2016 IEEE 22nd International Conference on Parallel and Distributed Systems

Enabling Mobile Device Coordination over Distributed Shared Memory

Maosen Huang, Hengfeng Wei*, Yu Huang
State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing 210023, China
csmaosenhuang@qq.com, hengxin0912@gmail.com, yuhuang@nju.edu.cn

***Abstract*—Distributed shared memory-based coordination has the advantage of simplifying the coordination logic to read-/write operations over the illusionary local memory. However, it is notoriously challenging to come up with a cost-effective implementation of the distributed shared memory. The implementation becomes more challenging in mobile environments, due to the resource constraints and the more rapid changes in the computing context. To this end, we propose the Mobile Distributed Shared Memory (MDSM) middleware to facilitate the development of mobile coordination applications. The key constructs in the shared memory are shared registers. Shared registers with different read/write patterns are implemented to facilitate flexible coordination. The registers also have different consistency semantics, to enable efficient tradeoff between data consistency and data access cost. An application framework is proposed to simplify the implementation of mobile coordination, relying on the middleware support from MDSM. A case study is conducted to demonstrate the usage of MDSM, where a soccer game application for the mobile phone is developed. Experimental evaluation is conducted to quantify different options of the consistency-latency tradeoff in the case study. The performance measurements show the cost-effectiveness of eventual consistency in this game. We also verify the read/write traces to further explain why eventual consistency practically performs better than it can guarantee.**

***Index Terms*—distributed shared memory, consistency-latency tradeoff, mobile coordination middleware, atomicity, eventual consistency;**

I. Introduction

Mobile coordination has been receiving more and more attention due to the widespread adoption of mobile smart phones and the significant advances in mobile computing/communication technologies[1][2]. Possible applications include multi-player mobile gaming, mobile social networking[3] and mobile phone-based brainstorming[4].

In many scenarios, the mobile coordination can be modeled as the concurrent update and timely synchronization of shared data over mobile devices (often mobile phones). This motivates the Distributed Shared Memory (DSM) based coordination[5]. The DSM exposes to the users a simple Read & Write interface[6][7]. The user accesses shared data in the illusion that the data is in its local memory. It is the implementation of the DSM which handles the data update and synchronization via message-passing primitives of network communication.

*Corresponding author.

The primary advantage of DSM-based coordination is that the upper-layer coordinating application can achieve better separation of concerns. Based on the support from the DSM, the coordination logic is greatly simplified to a collection of read/write operations. The developer can then focus on the application logic. This separation of concerns provides software engineering benefits in that it improves the intelligibility and maintainability of the application.

Though DSM-based coordination has important advantages, it is challenging to come up with a cost-effective implementation of the DSM, especially in mobile computing environments. Ensuring strong shared memory consistency can principally ease the development of upper-layer applications, but it also induces long latency to wait for the data updates. In contrast, weakly consistent data can be quickly accessed, but the burden of reasoning over stale data is transferred to the application developer. For example, in mobile gaming, often it is important to ensure quick response to operations from the user[8]. This can only be achieved at the cost of tolerating to certain degree stale data. In the dual example of collaborative editing, if the data being updated is of great importance, strong data consistency is necessary, and the users may be forced to wait.

The key problem in cost-effective implementation of the DSM is the tradeoff between data consistency and data access latency[9]. This problem is more challenging in mobile computing environments[10], since the mobile devices are often resource-limited. Moreover, the mobile devices are now equipped with various sensors, and may constantly sense changes in its computing context.

To address this challenge, we present Mobile Distributed Shared Memory (MDSM) - the DSM-based coordination middleware over mobile devices. The MDSM middleware consists of two essential parts:

- A variety of shared registers are implemented in the shared memory. The registers have different types of read/write patterns (e.g., Single Writer Multiple Reader, or Multiple Writer Multiple Reader) to enable flexible usage. More importantly, registers of different consistency models are implemented to enable efficient consistency-latency tradeoff.
- An application framework is proposed for the development of MDSM-based mobile coordination applications. According to the framework, the application is

1521-9097/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPADS.2016.16

constructed employing the Model-View-Control (MVC) pattern. The control component is in charge of the coordination, relying on the middleware support from the underlying MDSM.

A case study is conducted to demonstrate the usage of MDSM, where a mobile phone soccer game application is developed. The game control is based on the accelerometer of the mobile phone, and the game is modeled as the update of shared game data. Experimental evaluation is conducted to quantify different options of the consistency-latency tradeoff in the case study. The performance measurements show the cost-effectiveness of eventual consistency models in this game. We also verify the read/write traces to further explain why eventual consistency practically performs better than it can guarantee.

The rest of this work is organized as follows. Section 2 presents the design of MDSM. Section 3 presents the case study and Section 4 presents the performance measurements. Section 5 conclusions this work with a brief summary and discussions on the future work.

II. MDSM MIDDLEWARE

In this section, we present the structure of the MDSM middleware. First we present the overview of the MDSM middleware, then we discuss the DSM layer and the application layer in detail. The open source implementation can be found at *https://github.com/methor/MDSM*.

A. Overview of MDSM

Our MDSM middleware consists of three layers and a crosscutting module as illustrated in Fig. 1:

1) The bottom layer. This layer includes the sensor module and the network module. The network module adopts the standard TCP protocol and user-defined network topology. It reuses underlying socket resources to reduce garbage collecting of Java objects thus achieving energy-saving. The sensor module encapsulates the functionality of a variety of sensors. Sensor sampling generates *sensor events* that are sent to the application layer. These events contain newly-collected sensor data. Note that the sensor event frequency affects the performance and behavior of the upper application, thus it is a key parameter of sensors in this module. It can be controlled dynamically as the numbered arrow 6 in Fig. 1.
2) The DSM layer. This layer consists of user-defined *shared registers* defined in section II-B. This layer exposes a uniform `read/write` interface of shared registers to the upper layer. The upper layer coordinates with remote devices via invoking DSM `read/write` operations shown as numbered arrows 1 and 2 in Fig. 1. Each shared register is associated with a *consistency model* to achieve expected *consistency*. The DSM layer communicates with remote devices via network `send/receive` primitives shown as numbered arrows 7 and 8 in Fig. 1.
3) The application layer. This layer adopts the MVC (Model-View-Controller) pattern. The M-part is the data model of the application, the V-part visualizes the data model, and the C-part controls modifications on the data model. The controller interacts with the sensor module interface and the DSM layer interface. It periodically receives sensor events from sensor module and possibly changes the sensor event frequency. It reads from and writes to remote devices via the DSM layer's `read` and `write` operations, respectively.
4) The crosscutting logging module. The logging module keeps track of runtime information, such as the sensor event frequency of the sensor module, the latency of DSM operation, the latency of application controller operation, and the state of the application model. We use this module to monitor and analyze system states.

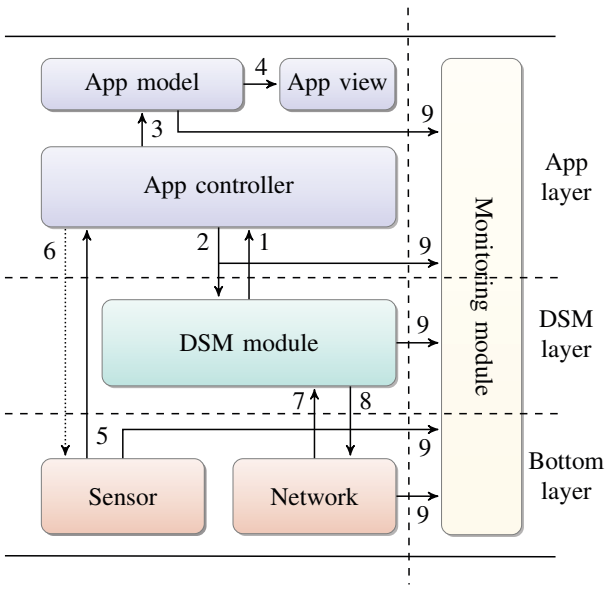

Fig. 1. Architecture of the MDSM middleware. Explanation of the numbered arrows. 1: App controller invokes DSM `read`; 2: App controller invokes DSM `write`; 3: App controller updates App model; 4: App screen takes data of App model to refresh screen; 5: sensor send sensor events to App controller; 6: App controller (possibly) controls sensor event frequency; 7 is the DSM asynchronously receive network packages; 8: DSM invokes network primitives to send network packages; and 9: monitoring module collects runtime information of different layers.

B. The DSM layer

Applications often coordinate via reading and writing to *shared variables*. The DSM actually maintains a replica of shared variables, which is continuously updated via synchronizing with replicas on remote devices using network `send/receive` primitives. The DSM abstracts the notion of *shared registers* from shared variables. Shared registers masks underlying network primitives and provides the DSM `read/write` interface to the application layer. Shared registers altogether consist of the overall *distributed shared memory* (DSM). Shared registers have two major properties: *access pattern* and *consistency model*.

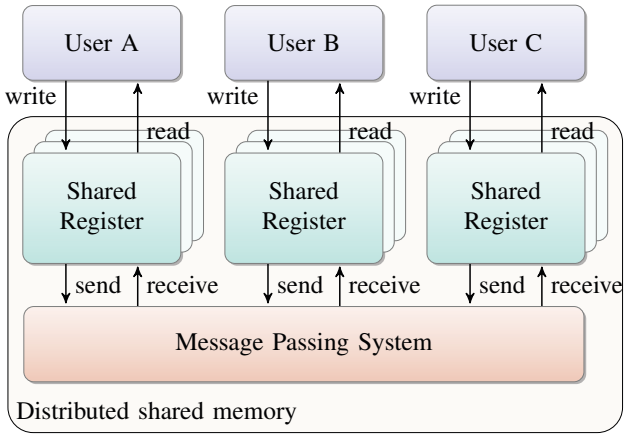

Fig. 2. DSM illustration

- *Access pattern*. The access pattern specifies the way shared registers are accessed. Typical access patterns include Single Writer Multiple Reader (SWMR) and Multiple Writer Multiple Reader (MWMR). In the former case, the register is owned by a particular device. It is only allowed to be written by the owner, while all devices can read the register. In the latter case, all devices can both read and write the register.
- *Consistency model*. Consistency models specify what values may be returned by a given read[11]. Consistency models can be strong or weak and consist the *consistency spectrum*. Here we briefly introduce several consistency models: atomicity, causal consistency, PRAM consistency, and eventual consistency. Atomicity provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response[12]. PRAM consistency requires only that the writes of each process be seen in *program order* at all other processes. Program order is defined as the sequential order of local operations. Thus, each process must sequence its own operations and the writes of other processes[13]. Causal consistency first defines the *write-into order*; that is, a read operation must obtain its value from only one write operation. Then the *causality order* is defined as the transitive closure of the program order and the write-into order. Causal consistency specifies that for every process, its own operations and all write operations from other processes conform to the causality order[13]. Eventual consistency is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value[14][15]. We have implemented the atomic consistency model, the causal consistency model, and the eventual consistency model. In section IV-D it will be explained that under particular circumstances, the eventual consistency model outperform what it guarantees. A significant difference is that the atomic consistency model *blocks* at every `read/write` invocation while the others have the *non-blocking* feature.

Intuitively, there is a tradeoff between strong consistency and low latency. Extra message passing rounds or blocking is needed to ensure high consistency thus increasing latency. Applications should choose an appropriate consistency model from the consistency spectrum.

C. The Application layer

We construct the application layer with the classic MVC pattern. The model component maintains the application state to be shown to end users via the view component. Note that the model here is different from the state of shared memory. The model component takes snapshots of shared registers in DSM as its states. The model component may stay untouched, but meanwhile the shared memory has been updated several times.

The view component simply refreshes the screen based on the state of the model component periodically. The view component starts a background loop thread to persistently retrieve the model state and redraw the screen.

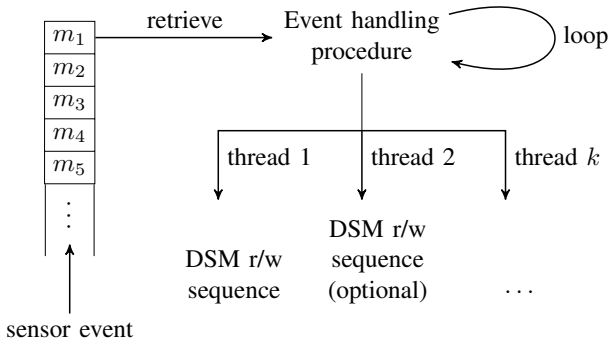

Fig. 3. Sensor event handling of the App controller

The controller component mainly consists of an event handling procedure that potentially starts several threads as shown in Fig. 3. It interacts with the DSM layer and the sensor module. The sensor module periodically generates event data and sends the event to the application controller component. The controller component puts the event at the tail of its internal event queue and picks the event at the head of the queue to handle. The event handling procedure contains a finite loop that handles an event in every iteration. Iterations in the event handling procedure adopts the same DSM `read/write` sequence on shared registers. The event handling procedure may start several threads and disseminate these operations to different threads. For example, the application has to coordinate with 5 remote devices, and it takes 5 operations to coordinate with each device. It sums up to 25 operations which may slow down the processing rate. Since all devices are identically treated, we can start 5 threads, each

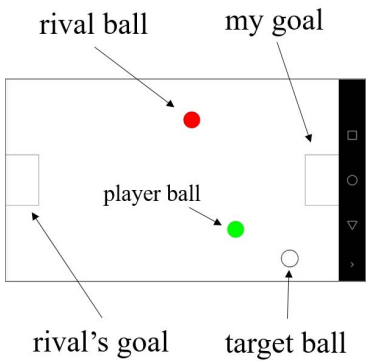

Fig. 4. Soccer game screenshot

invokes 5 DSM operations. Therefore the processing time of the event handling procedure is greatly reduced.

The controller component is also responsible to adjust the rate of sensor events, if needed. The rate of the controller's event handling procedure may not match the sensor event frequency. If the former is greater than the latter, the event handling procedure of the controller will starve. In contrast, the internal event queue of the controller will enlarge too fast to be handled by the event handling procedure. These scenarios could be alleviated by dynamically adjusting the rate of sensor events based on recent changes of the event queue's size.

III. Mobile Gaming: a Case Study

In this section we demonstrate a mobile multi-player soccer game developed upon MDSM. First we overview the game, then we demonstrate how to develop the game application based on MDSM.

A. Game Overview

In this case study, the game supports two players combat each other. Each player controls a ball. Besides the player balls, there is a target ball belonging to nobody. Players combat in a field. At every end of the field, there stands a goal belonging to one side. Players control his ball via the accelerometer of the device to hit the target ball into the rival's goal. The player scores once the target ball enters the rival's goal zone. The screenshot of the soccer game is shown in Fig. 4.

The players' devices share the same view of the field and balls. To enable view sharing, the application must continuously updates game state and transmits new changes to remote devices. Hard-coding the coordination process needs a number of asynchronous procedures and it is hard to maintain, refactor and reuse the code. Such implementation may achieve a certain consistency, however, when switching to a different scenario, the application may specify a different consistency. In other words, it is inflexible in different situations. We expect the implementation to be intelligible and provide different consistency models. These features are well supported by the DSM. The following section discusses how to build the game using MDSM.

B. Build the Game Using MDSM

We use the notion of `state = reg.read()` to indicate reading from the shared register `reg` into the variable `state`, and the notion of `reg.write(state)` to indicate write the content of variable `state` into the shared register `reg`. We abstract three registers from the game: one represents the target ball, each of the other two represents a player ball. The setup of shared registers and the read/write sequence are listed in Fig. 5.

Because the player ball is controlled exclusively by the player himself while the target ball can be modified by all players, different access patterns are applied to them. The MWMR pattern is applied to the register of the target ball and the SWMR pattern is applied to the registers of the player balls. We access them sequentially from a thread in the application controller module. By thinking of these accesses as *rounds*, we intuitively specify an order for them. Suppose player *B* is the rival of player *A*, and player *A* controls ball *A* and player *B* controls ball *B*. At the beginning of each round, player *A* wants to inform player *B* of his new movement before doing anything else, so `self.write(newSelf)` is performed. Then *A* wants to learn *B*'s new movement, so *A* invokes `newRival = rival.read()`. At this time, *A* knows both players' up-to-date movements. Then *A* invokes `newTarget = target.read()` to get the state of the target ball. After local computation, variables `newSelf`, `newRival` and `newTarget` are updated. Finally, *A* calls `write(newTarget)` to update the state of the target ball. Note that *A* will inform *B* of `newSelf` at the beginning of the next round.

IV. Performance Measurements

Since we focus on the consistency-latency tradeoff, both consistency and latency are measured for the atomic consistency model and the eventual consistency model. We characterize consistency with multiple *divergences* and latency with the *event waiting latency*. First we introduce experiment setup, then we separately present effects of tuning the sensor event frequency and the network delay, finally we give the conclusion of experiments and extra insights on eventual consistency.

A. Experiment Setup

We conduct our experiments on our ball game using two mobile phones. The model of mobile phones is HUAWEI P8-LTE ALE-CL00 and the operation system is Android 4.4.4. Connections between mobile phones are established by utilizing android P2P framework. We connect mobile phones with an Asus Zenbook UX305LA laptop for clock synchronization.

We measure consistency and latency of the atomic consistency model and the eventual consistency model against two varying arguments: *sensor event frequency* and *upper bound of injected delay*. The notion of sensor event is mentioned

```
/* DSM registers setup, 1st param is
   access pattern, 2nd is consistency
   model */
SharedRegister self = new SharedRegister(
   SWMR, atomicity);
SharedRegister rival = new SharedRegister
   (SWMR, atomicity);
SharedRegister target = new
   SharedRegister(MWMR, atomicity);

/* DSM r/w sequence */
Ball newSelf, newRival, newTarget;
newSelf = updateFromSensor();
// seq operation 1
self.write(newSelf);
// seq operation 2
newRival = rival.read();
// seq operation 3
newTarget = target.read();
// update newSelf, newRival, and
   newTarget
computeNewState(newSelf, newRival,
   newTarget);
// seq operation 4
target.write(newTarget);
```

Fig. 5. Example of DSM setup and r/w sequence for one player

in II-A. The injected network delay is distributed randomly and uniformly of every TCP write operation within the upper bound. There are several kinds of latency that can be identified. The first one is the *processing latency*, which indicates the elapsed time from the start to the end of an iteration of the event handling procedure. We measure the *event waiting latency* that is closely related to the processing latency. It denotes the waiting time from the time instant when the sensor event is generated to the time instant when the sensor event is actually handled. Note that the event waiting latency also increases as the processing latency increases. For our application, consistency is characterized by *divergences* between devices. There are three kinds of divergence: acceleration divergence, speed divergence and position divergence. These divergences are measured at the same *global time*. The position divergence indicates the difference of positions of balls on different devices. Likewise, the speed divergence indicates the difference of speeds of balls on different devices and the acceleration divergence indicates the difference of accelerations of balls on different devices. Due to the limit of space we discuss the position divergence and the acceleration divergence, since the result of the speed divergence is similar to that of the position divergence.

In each experiment, there are 10000 event handling records on each device. We repeat every experiment 3 times and take the average. Measuring the event waiting latency is straightforward, yet measuring divergences needs more explanation. Measuring divergences requires a global time over all devices. Usually it is achieved by synchronizing with the same PC. In practice, however, the latency of communication with PC reaches to 40ms which affects the performance of consistency models. We tackle this problem by an intuitive heuristic: the time elapsing rate is about the same on every device. So we synchronize with PC at the first time, and then set this PC time as the global time. The global time is updated locally; that is, the global time is added by the elapsed time since the latest update. Snapshots of the game model are taken at the end of every round of the DSM read/write sequence. Meanwhile, they are tagged with global time and then logged in a file. We use snapshots in the logging file to compute divergence. Because execution rates on separate devices differ, their logging timing may not match. We need to compare snapshot records at the same global time. To this end, we use the interpolation method to estimate snapshot records at specified global time instants. Tagged global time instants in snapshot records serve as the interpolation argument. Data on every other dimension, such as coordinates along the x-axis or speed along the y-axis is treated as the interpolation value. After the interpolation processing, we combine these dimensions so that the artificial snapshot records are comparable. We compute the weighted average of Euclid distances of balls as the position divergence. Since the target ball is considered more important, the weight of Euclid distance of the target ball is multiplied by 2. The same evaluating method is applied to the speed divergence and the acceleration divergence. Finally, the normalization step is taken so that the maximum value of these divergences is 1.

B. Effects of Tuning the Sensor Event Frequency

Fig. 6 shows how the acceleration divergence changes when the sensor event frequency increase. As the frequency increases, We find that the acceleration divergence of the eventual consistency model grows monotonously while that of the atomic consistency model almost stays unchanged. Recall that the atomic consistency model is *blocking*. The atomic consistency model is insensitive to the sensor event frequency because it handles the sensor event about 10 times per second at most. It is calculated by the fact that a round of DSM read/write sequence takes around 100ms to proceed. The acceleration divergence of the eventual consistency model increases because it is more likely that values on shared registers may be randomly overwritten before retrieval when the frequency increases.

A different pattern is found as for the position divergence. Fig. 7 shows how the position divergence changes when the sensor event frequency increases. The eventual model performs better in contrast to the result of the acceleration divergence. The difference is that positions of balls are calculated with time interval involved. For the eventual consistency model, time interval is affected by sensor event frequency. The increase of frequency is equivalent to multiplying a factor smaller than 1 to the position divergence thus reducing the

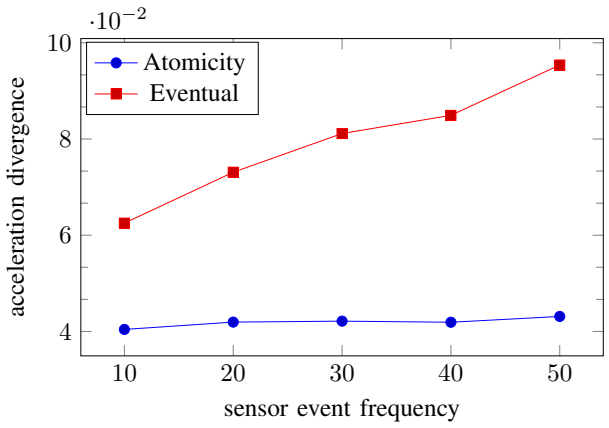

Fig. 6. Acceleration divergence against sensor event frequency

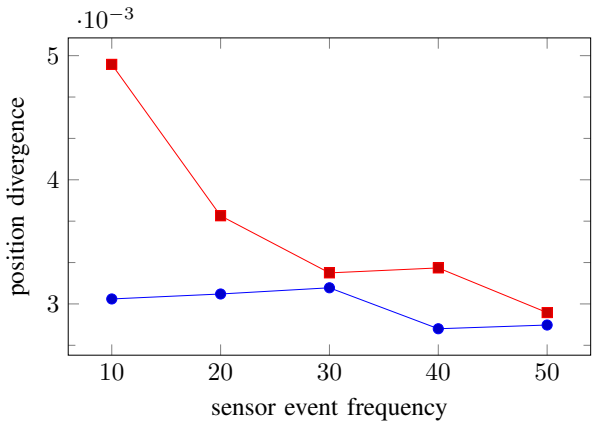

Fig. 7. Position divergence against sensor event frequency

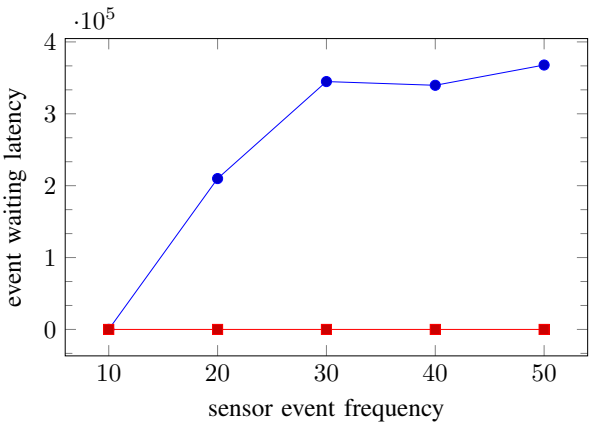

Fig. 8. event waiting latency against sensor event frequency

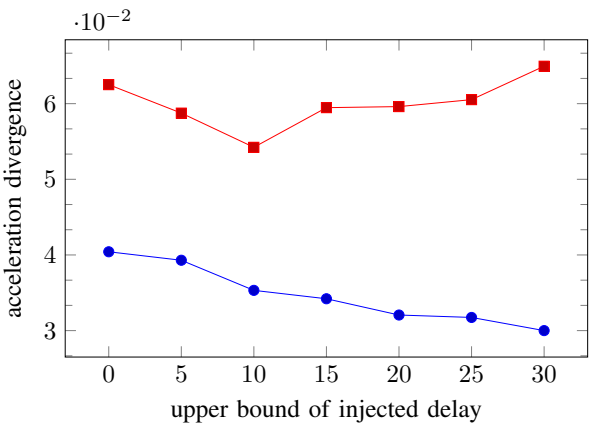

Fig. 9. Acceleration divergence against the upper bound of injected delay

position divergence. The atomic model performs stable, as expected.

Fig. 8 shows how the event waiting latency changes when the sensor event frequency increases. The event waiting latency remains slightly greater than 0 for the eventual consistency model, while it increases significantly for the atomic consistency model. The reason why the latency increases quite fast for the atomic consistency model is that the rate of event handling is slower than sensor event frequency. In this situation events are stacked in the event queue of the application controller so that more waiting time will be consumed before the events are handled as the sensor event frequency increases.

C. *Effects of Tuning the Network Delay*

Fig. 9 shows how the acceleration divergence changes when the upper bound of injected delay increases under the condition that the sensor event frequency takes the value of 10. The reason why we choose this frequency is that it ensures fairness for comparing two consistency models. The acceleration divergence of atomic consistency model decreases slowly because the rate of event handling drops as the upper bound of injected delay increases. The eventual consistency model stays stable. Because most network messages are received within 100ms, there is little difference whether injecting delay or not.

Fig. 10 shows how the position divergence changes when the upper bound of injected delay increases under the condition that the sensor event frequency takes the value of 10. The eventual consistency model performs stable while the position divergence of the atomic consistency model increases approximately linearly. In the case of the atomic consistency model, the rate of event handling drops as the delay increases, which causes greater time intervals to be used in computing position thus enlarging position error. For the eventual consistency model, because the acceleration divergence is stable, the position divergence is stable as well. The atomic consistency model performs better than the eventual consistency model without injected latency, but surpassed by the eventual consistency model as the injected delay increases.

Fig. 11 gives the event waiting latency against the upper bound of injected delay. For the eventual consistency model, the latency remains slightly greater than 0 as in the case of without injected delay. For the atomic consistency model, however, the latency increases faster than the latency without

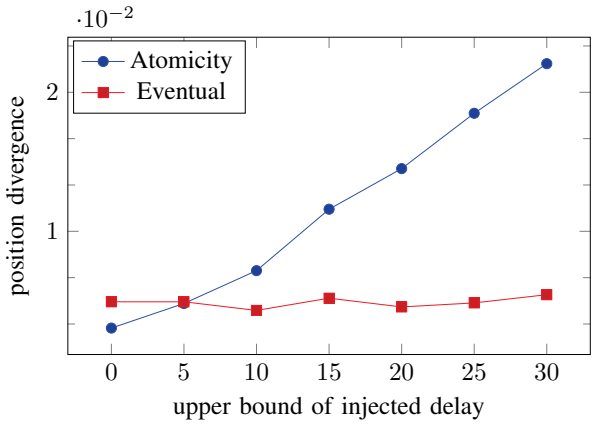

Fig. 10. Position divergence against the upper bound of injected delay

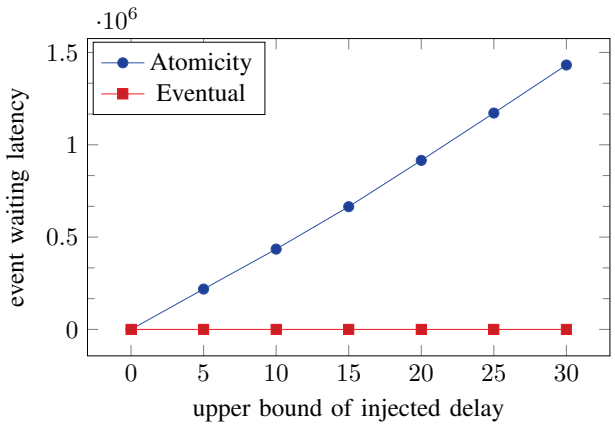

Fig. 11. Event waiting latency against the upper bound of injected delay

injected delay. Note that the scales on the y-axis in Fig. 8 and Fig. 11 are different. From Fig. 10 and Fig. 11, we find that the atomic consistency model is sensitive to network delay while the eventual consistency model is not.

D. Lessons Learned

Experiments above demonstrate the effects of tuning the sensor event frequency and the network delay. The result of acceleration divergence shows that the atomic consistency model outperform the eventual consistency model in all cases, and the eventual consistency model becomes more inconsistent as the sensor event frequency increases due to overwritten on shared registers. However, it shows that the latency of the eventual consistency model is significantly lower than that of the atomic consistency model thus improving interactive responsiveness. Furthermore, the feature of low latency improves metrics that involve calculation using time intervals, such as the position divergence in our case study. Finally, the atomic consistency model shows great sensitiveness to the network delay since the position divergence increases when tuning the network delay. In contrast, the eventual consistency model is insensitive to the network delay. In summary, the eventual consistency model has much lower latency than that of the atomic consistency model, and it is only slightly weaker than the atomic consistency model in terms of consistency.

We investigate the reason why the eventual consistency model behaves much better than it can guarantee via experiments. We collect 10 running traces of the case study, each consisting of 8000 read/write operations. The running traces is verified against PRAM consistency. The verification results show that all running traces conform to PRAM consistency. PRAM consistency is practically guaranteed since the underlying TCP protocol guarantees ordered receive the same as the order of sending messages. Such property guarantees that write operations of every individual device are observed following the same order by all devices, which is exactly what PRAM consistency requires.

Moreover, our implementation of the eventual consistency cannot guarantee causal consistency. However, we find by experiments that only a quite limited percentage of read operations violates the semantics of causal consistency. Specifically, we conduct 10 experiments each containing 4000 read operations. Only 1.12% out of 40000 operations violates causal consistency. It is also the message ordering actually provided by the TCP protocol which makes the traces "almost" causally consistent.

As for energy efficiency, we highly reuse underlying socket resources, which reduces garbage collecting of Java objects and thus saves energy. Since the soccer application frequently interacts with each other, the saved energy is insignificant compared to the energy consumed by frequent network communication.

V. Related Work

From the perspective of this work, we discuss two types of related work: mobile coordination middleware and DSM-based middleware.

Examples of mobile coordination include LIME[16], TOTA[17] and MIPA[18]. LIME and TOTA build tuple spaces among mobile devices. Shared tuple spaces support generative communication among computing entities. The communication is generative since the tuple generated by certain entity has an independent existence in the tuple space. The computing entities are thus highly decoupled. Each entity participates in the coordination only by accessing data, without knowing or interacting with other entities. Though such generative communication greatly simplifies the programming of coordination, it imposes communication cost on the underlying network system. Tuple space does not enable flexible tuning of access pattern of data tuples. Nor does it facilitate flexible consistency-latency tradeoff. The MIPA middleware focuses on coping with the intrinsic asynchrony among mobile devices. Global predicates are detected over the asynchronous traces of mobile device interaction. Using MIPA, the mobile devices principally publish all its traces to a centralized middleware, while directed coordination among mobile devices are not supported.

Examples of DSM-based coordination include IVY[19] and object-based DSM middleware for Java[20]. IVY implements DSM on a local ring network. It achieves to migrate data and process between processors in a distributed system without violating memory coherence. The object-based DSM middleware utilises Message Oriented Middleware (MOM) to provide reliability and scalability for Java shared objects. The middleware works even when the channels are unreliable and the packets can be lost.

The MDSM middlware we propose in this work employs distributed shared memory to support flexible coordination. We focus on providing rich shared memory access patterns and facilitating efficient consistency-latency tradeoffs, to tackle the challenges from the mobile coordination environment.

VI. Conclusion and Future Work

In this work, we present Mobile Distributed Shared Memory (MDSM) - the DSM-based coordination middleware over mobile devices. To enable efficient consistency-latency tradeoff, the MDSM middleware implements a variety of shared registers of different types of read/write pattern and different consistency models. An application framework is proposed for the development of MDSM-based mobile coordination applications. A case study is conducted and evaluated to demonstrate the usage of MDSM and show the cost-effectiveness of the MDSM middleware.

In future work, we will need to improve scalability of the MDSM middleware. We will also adopt the MDSM middleware to the mobile/cloud computing environment. More experiments and case studies will need to be conducted based on the MDSM middleware.

Acknowledgements

This work is supported by the National 973 Program of China (2015CB352202) and the National Science Foundation of China (61272047, 91318301, 61321491).

References

[1] G.-C. Roman, A. L. Murphy, and G. P. Picco, *Coordination and Mobility*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 253–273. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-04401-8_10

[2] I. Rahwan, F. Koch, C. Graham, A. Kattan, and L. Sonenberg, "Goal-directed automated negotiation for supporting mobile user coordination," in *International and Interdisciplinary Conference on Modeling and Using Context*. Springer, 2005, pp. 382–395.

[3] N. Jabeur, S. Zeadally, and B. Sayed, "Mobile social networking applications," *Commun. ACM*, vol. 56, no. 3, pp. 71–79, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2428556.2428573

[4] I. R. N. C. Karunatillake, N. R. Jennings and T. Norman, "Argument-based negotiation in a social context," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS*, 2005, pp. 1331–1332.

[5] X. Jiang and Y. Huang, "Cbbr: Enabling distributed shared memory-based coordination among mobile robots," in *7th Asia-Pacific Symposium on Internetware*, 2015.

[6] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[7] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *IEEE Parallel Distrib. Technol.*, vol. 4, no. 2, pp. 63–79, Jun. 1996. [Online]. Available: http://dx.doi.org/10.1109/88.494605

[8] J. O. B. Soh and B. C. Y. Tan, "Mobile gaming," *Commun. ACM*, vol. 51, no. 3, pp. 35–39, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1325555.1325563

[9] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1109/MC.2012.33

[10] Y. Huang, J. Cao, B. Jin, X. Tao, J. Lu, and Y. Feng, "Flexible cache consistency maintenance over wireless ad hoc networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 8, pp. 1150–1161, Aug. 2010. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2009.168

[11] R. C. Steinke and G. J. Nutt, "A unified theory of shared memory consistency," *Journal of the ACM (JACM)*, vol. 51, no. 5, pp. 800–849, 2004.

[12] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[13] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[14] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[15] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5. ACM, 1995, pp. 172–182.

[16] A. L. Murphy, G. P. Picco, and G.-C. Roman, "Lime: A coordination model and middleware supporting mobility of hosts and agents," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 3, pp. 279–328, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151695.1151698

[17] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: The tota approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 4, pp. 15:1–15:56, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1538942.1538945

[18] Y. Yang, Y. Huang, X. Ma, and J. Lu, "Enabling context-awareness by predicate detection in asynchronous environments," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 522–534, 2016.

[19] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989. [Online]. Available: http://doi.acm.org/10.1145/75104.75105

[20] M. Mazzucco, G. Morgan, F. Panzieri, and C. Sharp, "Engineering distributed shared memory middleware for java," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2009, pp. 531–548.