Available online at www.sciencedirect.com

SciVerse ScienceDirect

Physics Procedia 25 (2012) 700 – 707

Physics Procedia

2012 International Conference on Solid State Devices and Materials Science

Decentralized Checking Context Inconsistency in Ubiquitous Mobile Computing Environments[1]

Daqiang Zhang[1,2], Zhijun Yang[1,2], Hongyu Huang[3], Qin Zou[4]

[1]School of Computer Science, Nanjing Normal University
[2]Jiangsu Research Centre of Information Security & Confidential Engineering
[3]Department of Computer Science, Chongqing University
[4]School of Remote Sensing and Information Engineering, Wuhan University

Abstract

Contexts are often noisy in ubiquitous mobile computing environments due to user mobility, unreliable wireless connectivity and resource constraints. Various schemes have been proposed to check context inconsistency for ubiquitous mobile applications. However, most of them require central control. This requirement inhibits their working in ubiquitous mobile environments, which are characterized by the asynchronous coordination among computing devices. In this paper, we propose DCCI scheme – Decentralized Checking Context Inconsistency for ubiquitous mobile applications by exploiting the preference-based locality that denotes context inconsistency occurs among the nodes that impose various restrictions on the same context. According to this locality, DCCI constructs a preference-based shortcut structure to check inconsistency within shortcuts. Extensive experiments show that DCCI can accurately and efficiently check context inconsistency in a fully distributed manner.

1. Introduction

Ubiquitous mobile computing allows users to share services without knowing underlying technologies by seamlessly integrating networked computing devices with users and their ambient environments [1, 2, 3]. This is mainly accomplished by context-awareness, assisting mobile applications in automatically adapting to changeable contexts [4]. Contexts, including application requirements, device capability, user location and behaviors, and relationships among users, denote the pieces of information that capture the characteristics of ubiquitous mobile computing environments. Contexts can be gathered by various devices, such as hand-held devices, wireless sensors and RFID.

[1] This work is supported by the National Natural Science Foundation of China (Grant Nos. 61073118 and 61003247).

 doi:10.1016/j.phpro.2012.03.146

Contexts are inherently imprecise and noisy so that context inconsistency frequently occurs, e.g., different observations for the same temperature from two sensors, and contradictions in a computation task's context [5, 6, 7]. This is partially because sensor technology is prone to error. The sources of errors consist of, but are not limited to, inaccuracies in the measurement and noise from internal components. This is also because contexts are highly complex. For instance, in asynchronous and heterogeneous mobile environments, contexts easily become obsolete and dynamically vary with users. Detailed reasons for inevitably noisy contexts for ubiquitous mobile computing can be found in [8].

2. DCCI: a decentralized scheme for checking context inconsistency

In this section, we introduce DCCI: a Decentralized Scheme for Checking Context Inconsistency. We first describe our system model. Then we present the detail design of DCCI, followed by discussions.

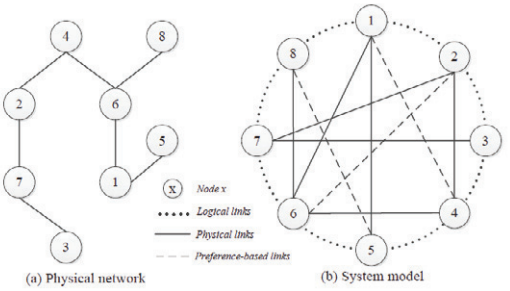

Fig. 1. Illustration of the system model of DCCI, which is constructed by creating shortcuts among nodes that impose diverse constraints on the same context.

2.1System Model

The design philosophy of DCCI is to seek a simple, robust, fully distributed and scalable system that can efficiently check context inconsistency. DCCI is inspired by a simple and efficient principle called preference-based locality, denoting that the nodes taking advantage of the same context can check constraints over various context instances. These nodes, preferring the same context with distinct constraints on context attributes, are called *pNeibor* nodes and they form a preference-based list named *pNeiborLst*. In Figure 1(b), *pNeibor* nodes 5 and 8 impose different constraints on the same context and thus they comprise a preference-based list *pNeiborLst*, while *pNeibor* nodes 1 and 4 add constraints on a type of contexts and form another *pNeiborLst*. Suppose a node in a *pNeiborLst* collects a context, it and its neighbors in the *pNeiborLst* can rapidly check whether context inconsistency does exist or not by evaluating their constraints on context properties, respectively. Intuitively, by exploiting the preference-based locality, DCCI assists pervasive applications in remarkably narrowing the checking scope of context inconsistency detection, rather than broadcasting over the entire network. Meanwhile, DCCI specifies the targeted destinations to which the being checked contexts should be disseminated. Which significantly reduces the communication overhead and accelerates the context dissemination and checking. Given the various requirements from every node, DCCI is supposed to provide a function for checking diverse constraints on contexts. For instance, we assume the location of a user at the specified time should be unique, i.e., the location of this user can not be totally different at the same time snapshot. As a result, the problem of context inconsistency checking turns out to the maintenance of the preference-based locality.

In DCCI, pervasive computing applications are modeled as a loosely-coupled distributed system without any central control or shared memory. All nodes are equivalent and may frequently switch scenarios, i.e., joining or leaving a pervasive network. Without loss of generality, we assume that all nodes (participating devices and users) are uniformly distributed in a pervasive space where they can independently move during a finite period of time *t* with a speed v randomly chosen in the interval $0 \leqslant v \leqslant 2m/s$ (the upper bound is set according to the average human walking speed that is about 1.3m/s) in arbitrary directions to reflect user displacements. At the end of the period t, a node may stay, leave or move on. Note that the speed of nodes can be altered if necessary so that these nodes may continue their movements until reaching a border. These nodes communicate by message-passing to form an overlay. Communications suffer from finite but unbounded message delay, and all communications are directional (i.e., unidirectional communication can be detected and hidden at the network layer).

Figure 1 illustrates the system model of DCCI, consisting of three types of links: physical links from the physical network, logical links from the underlying the network overlay, and shortcuts from the shortcut structure that is built on top of the network overlay. Physical links denote that nodes can communicate directly, whereas logical links and the shortcut structure represent nodes can communicate logically. In order to construct the structure, i.e., linking *pNeibor* nodes that are in the same *pNeiborLst*, DCCI builds an overlay, identifies nodes that are associated with the *pNeibor* relationship, and then discovers shortcuts by the preference-based locality.

2.2DCCI's Steps

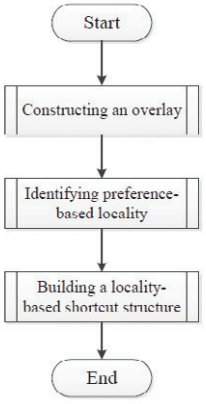

Fig. 2. The detailed design of DCCI, involving three steps – constructing an overlay, identifying preference-based locality and building a locality-based shortcut structure.

Figure 2 illustrates the DCCI scheme, which checks context inconsistency by three steps – constructing an overlay, identifying preference-based locality and building locality-based structure. In addition, DCCI provides a function to maintain the network under frequent node churns (e.g., node joining and leaving). In Sections II-B1, II-B2, II-B3, we will detail these steps. Meanwhile, DCCI provides a maintaining service in Section II-B4. It is worth emphasizing that DCCI is compatible with most existing overlays, such as Chord, CAN and Gnutella. Pervasive applications is capable of customizing the locality and this locality based structure in light of their requirements. Thus, DCCI demonstrates that the locality-based structure may be a performance enhancement.

1) Constructing an overlay: This step aims to construct an overlay by organizing all the nodes in a logical ring, which is adapted for pervasive computing applications to locate the source without flooding in the network.

Each node and key assign themselves through a consistent hashing function such as SHA algorithm as well as its various variants. The identifiers for nodes and keys are generated by hashing node addresses and keys, respectively. Thus, the keys are mapped to the overlay and handled by their nearest nodes on a logical ring. Each node keeps track of a successor and a predecessor all the times, thus building a logical overlay.

Concurrent node churns (i.e., nodes join and leave concurrently) occur frequently due to unpredictable mobility, resource limitation and unreliable wireless connection in pervasive computing environments characterized by asynchronous coordination among computing devices. The lack of consideration of concurrency and device heterogeneity inhibits the effectiveness of most existing schemes from peer-to-peer and network coding fields. On the other side, concurrent churns can be handled by lock mechanisms, which definitely incur many communication overheads. Given that the targeted applications are not real-time systems, DCCI leverages the underlying overlay to capture and reflect the node churn, which guarantees the best performance of context consistency checking in most cases.

Suppose a node n joins the overlay network in DCCI, and ID is between nodes n_s and nt. Nodes n, n_t and n_s initiate an joincheck process, respectively. At the beginning, node n sets n_t and ns as its successor and predecessor. Then, node n checks whether node ns set it as a successor. If not, node n will notify ns. Node n does the same process for node nt. Finally, the system reaches a stable and correct status. However, due to network latency as well as other failures, the above step may not be correctly executed. Consequently, the joining node will periodically check affected nodes to update their information (once in DCCI). The pseudo-code of concurrent join operation is given as Algorithm 1.

Algorithm 1 Concurrent join operation in DCCI

```
1:  procedure JOIN
2:      // node n joins the network and
3:      // its ID is between nodes n_s and n_t
4:      DCCI.join(n)
5:          DCCI.setPrecessor(n)=n_s;
6:          DCCI.setSuccessor(n)=DCCI.getSuccessor(n_s);
7:          DCCI.setPrecessorState(n)=false;
8:          DCCI.setSuccessorState(n)=false;

9:      // node n periodically checks the routing information
10:     // of nodes n_s and n_t
11:     while (DCCI.setPrecessorState(n) &&
12:             DCCI.setSuccessorState(n)) do
13:                 if DCCI.getSuccessor(n_s).inequal(n)
14:                     DCCI.notify(n->n_s);
15:                 else DCCI.setSuccessorState(n)=true;
16:                 if DCCI.getPrecessor(n_t).inequal(n)
17:                     DCCI.notify(n->n_t)
18:                 else DCCI.setPrecessorState(n)=true;
19:     end while
20: end procedure
```

When a node leaves a specific pervasive network, it is committed to notify its predecessor and successor, who will update their information correspondingly. Owing to the unexpected failures and exceptions such as device failures and network latencies, all nodes have to periodically check their neighbors.

2) Identifying preference-based locality: Based on the above network overlay, DCCI aims at identifying the preference-based locality. The simplest method is broadcasting the preferences of every node over the entire network, which causes a huge number of communication overheads. Most forwarding algorithms reduces the cost associated with flooding the network by forwarding only to good

relays. However, it is difficult to decide whether an encountered node is good relay at the moment of encounter.

In order to successfully identifying preference-based locality at low costs, DCCI extends the delegation forwarding protocol by limiting the number of forwarding. The extended protocol helps a node to only forward a message to nodes with quality greater than any observed nodes so far for its message. In DCCI, it dramatically reduces the communication overheads.

3) Building a locality*-based shortcut structure:* In this section, DCCI intends to create a shortcut structure by discovering shortcuts among nodes that impose constrictions on the same context, i.e., linking the *pNeibor* nodes in the same *pNeiborLst* and then checking context inconsistency among the nodes in the *pNeiborLst*.

When a node n joins the system, it may have no idea about other nodes' preference in context requirements. It joins the underlying overlay network by hashing its address using consistent hashing, and certain keys previously assigned to the *n's* successor now become assigned to the *n*. Then, node n checks other nodes' context requirements with its requirements by searching over the underlying network. Once a *pNeibor* node *n'* is located, node n will ignore the reply from any other *pNeibor* nodes. It will copy the *pNeiborLst* of node *n'* , create shortcuts with related *pNeibor* nodes and notify them to add it in their respective *pNeiborLsts*. Thus, nodes in *pNeiborLst* know their *pNeibor* nodes' location and subsequent context consistency checking go through the *pNeibor* nodes with known addresses in the specific *pNeiborLst*. If a node cannot find *pNeibor* nodes, it will issue a request to the underlying overlay network. Upon a context conflict is detected, the detecting node will immediately notify the dependent applications to deal with this conflict. The pseudo-code of constructing a shortcut structure is illustrated as Algorithm 2.

Algorithm 2 Shortcut discovery in DCCI

```
1:  procedure JOIN
2:      // node n joins the network
3:      DCCI.Shortcut.join(n)
4:          // cp refers to context patterns
5:          DCCI.Shortcut.route(n.cp(requirements));
6:          if (DCCI.Shortcut.getFirstReply(n)=n')
7:          {
8:              DCCI.Shortcut.setpNeiborLst(n)−
9:           DCCI.Shortcut.clone(n'.pNeiborLst());
10:             DCCI.Shortcut.notify(n->
11:                   node m in n'.pNeiborLst());
12:             DCCI.Shortcut.addShortcutLst(n, m);

13:             // pNeibor nodes update their information
14:             for (m in n'.pNeiborLst()) do
15:                DCCI.Shortcut.updatepNeiborLst(m);
16:                DCCI.Shortcut.updatepShortcutLst(m);
17:             end for
18:         } else
19:             DCCI.route(n.cp(requirements));
20: end procedure
```

In DCCI, the locality-based structure is just a performanceenhancement. If context consistency can be checked in *pNeibor* nodes within a specific *pNeiborLst*, it can always be checked in the underlying overlay network. Moreover, the overlay can also detect some kinds of context inconsistency that cannot be detected by the locality-based structure. For example, two nodes that are located remotely and impose two different constraints on the context of Joanne' location – unique and redundant. At this time, DCCI does not incorporate these two nodes into its locality-based structure such that DCCI can not detect the location inconsistency.

4) Maintenance under node churns: In pervasive applications, frequent node churn leads to the change in network topology. Therefore, the shortcut structure must be adapted dynamically. The adaptation in the underlying overlay is discussed in Section II-B1, and thus this section will concentrate on the adaptation in the shortcut structure.

In DCCI, each peer continuously keeps track of its shortcuts' performance and updates its shortcut ranking. Once it fails, all of its assigned keys are reassigned to its successor. Any other keys and their respective assigned nodes' location remain unchanged. In the shortcut structure, at least one of the neighbors of the failure nodes will detect the failure and notify the others to adapt. Thus, the shortcut structure is kept up-to-date.

With respect to concurrent operations in the physical network, DCCI has to spend much time adapting the underlying overlay. In order not to increase the maintenance burden at the busy time, DCCI defers the maintenance of the shortcut structure.

3. Evaluation

In order to evaluate whether the proposed scheme is appealing for context consistency checking in pervasive computing environments, we carried out a series of experiments. We select success rate to measure the accuracy of checking context inconsistency, which is defined as a percentage of successfully detecting the inconsistency among all context inconsistency.

In the following, we first describe the experimental settings, and then analyze the evaluation results.

3.1Experimental settings

We evaluated DCCI over OMNet++, which is an extensible, modular, component-based C++ simulation library and framework for communication networks, queuing networking and performance evaluation (See http://www.omnetpp.org/ for detail information). We ran experiments in Windows XP (SP3) with 2.0 GB memory and 2.4 GHz CPU, selected averaged values over ten times as results and selected the ideal flooding scheme (i.e., the scheme works without influence of noise, congestion and latency) as our benchmark. Note that we are also developing a prototype over the multi-campus to evaluate the validness of DCCI in practice. We currently deploy various sensors, RFID, bluetooth into the environment and asks participants to randomly move.

In accordance with the presentation in Section II-A, we randomly generated 500, 2,000 and 5,000 static and mobile nodes as E500, E2000 and E5000, respectively. We assume that at most 8% of all nodes impose different constraints on the same context, i.e., context inconsistency occurs within less than 10% of all nodes. The characteristics of experiment configurations are listed in Table1.

Table 1 statistical feature of experiment configurations

Item/Configuration	E500	E2000	E5000
No. of nodes	500	2,000	5,000
No. of static nodes	100	500	1,500
No. of types of contexts	200	1,000	2,000
No. of context constrictions	200	800	1,600

In our experiments, the overall performance of DCCI is evaluated by the success rate, which is defined as the ratio that the successfully checked inconsistency to the total number of context inconsistency.

3.2Success rate

We first conducted several experiments over E500, E2000 and E5000 to check the overall success rate, and then to check the performance of preference-based shortcuts in success rate.

Figure 3 illustrates the averaged results of success rate over different experiment configurations for each inconsistency detection. The x axis is the success rate and the y axis is the sample time when the observation was made. The averaged success rate for E500 and E2000 is as high as 80% – 97%, although it decreases in E5000 when the numbers of nodes, contexts and context constrictions increase remarkably. The results show that DCCI achieves the high levels of success rate in checking context inconsistency, which indicates that DCCI slightly affects the checking accuracy compared with the ideal flooding scheme. The differences between DCCI and the flooding scheme are mainly caused by network latency, nodes' movement and unreliable connection.

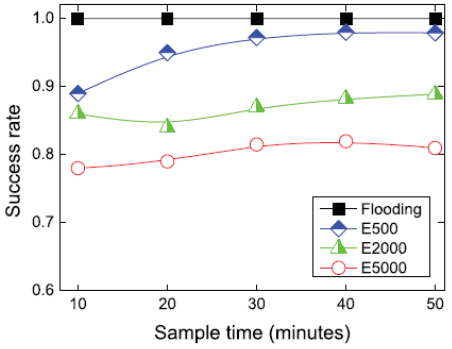

Fig. 3. The overall success rate over E500, E2000 and E5000 experiment configurations

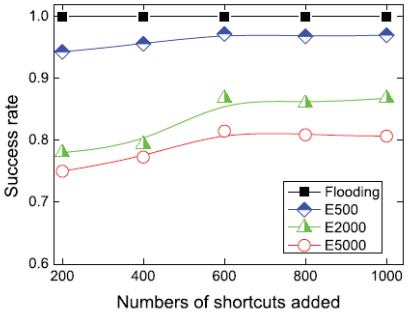

Fig. 4. Number of shortcuts added in context consistency checking over E500, E2000 and E5000 experiment configurations

Figure 4 illustrates how much the preference-based locality affects the averaged success rate, indicating the more shortcuts added, the better success rate that DCCI can achieve. The horizontal axis is

the number of shortcuts added during the sample time, and the vertical axis is the averaged success rate. With the growth of the number of shortcuts added, DCCI is able to obtain the higher levels of checking accuracy. After the number of the shortcuts added is about 600, DCCI achieves the best success rate. While the number of the shortcuts continues increase, the success rate diminishes. This is partially because the success rate is also dramatically affected by various environmental factors as it does in Figure 3. This is also because that the sizes of *pNeibor* nodes and *pNeiborLst* groups lead to a great amount of communication overhead as well as latency, alleviating the performance of preference-based locality.

6. Conclusion

In this paper, we have studied context inconsistency checking without central control in pervasive computing environments. Toward this objective, we have proposed DCCI: a scheme for Decentralized Checking Context Inconsistency, which checks context inconsistency by evaluating the constraints on the certain type of context instances and patterns over a shortcut structure. In order to construct the structure, DCCI first builds a simple overlay network and then leverages a preference-based locality. DCCI is a promising scheme for pervasive applications because it introduces a shortcut mechanism based on locality for performance enhancement. DCCI exploits the preference-based locality that nodes requiring the same context can check the inconsistency on this type of contexts. This locality can be tailored to according to the application requirements so as to achieve application goals.

However, DCCI currently suffers from several problems. We will investigate how to further reduce the message complexity during the maintenance of the underlying overlay network and the shortcutbased layer. We will also study how many shortcuts should be created such that DCCI can achieve best performance in terms of accuracy and efficiency for checking context inconsistency. Finally, we will study how to detect and interpret concurrent contexts in asynchronous and dynamic pervasive computing environments.

References

[1] M. Romn, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, “A middleware infrastructure for active spaces,” IEEE Pervasive Computing, vol. 1, no. 4, pp. 74–83, 2002.

[2] M. Weiser, “The computer for the 21st century,” Scientific American, vol. 265, pp. 66–75, 1991.

[3] W. Xue, H. Pung, P. P. Palmes, and T. Gu, “Schema matching for context-aware computing,” in Proc. of the 11th Intl. Conf. on Ubiquitous Computing, 2008, pp. 292–301.

[4] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, “The anatomy of a context-aware application,” in Proc. of the 5th Annual ACM/IEEE Intl. Conf. on Mobile Computing and Networking, 1999, pp. 59–68.

[5] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, “Heuristicsbased strategies for resolving context inconsistencies in pervasive computing applications,” in Proc. of the 28th IEEE Intl. Conf. on Distributed Computing Systems, 2008, pp. 713–721.

[6] H. Lu, W. Chan, and T. Tse, “Testing pervasive software in the presence of context inconsistency resolution services,” in Proc. of the 30th Intl. Conf. on Software Engineering, 2008, pp. 61–70.

[7] S. R. Jeffery, M. Garofalakis, and M. J. Franklin, “Adaptive cleaning for rfid data streams,” in Proc. of the 32nd Intl. Conf. on Very Large Data Bases, 2006, pp. 163–174.

[8] E. Elnahrawy and B. Nath, “Cleaning and querying noisy sensors,” in Proc. of the 2nd ACM Intl. Conf. on Wireless Sensor Networks and Applications, 2003, pp. 78–87.