

Decentralized checking of context inconsistency in pervasive computing environments

Daqiang Zhang · Min Chen · Hongyu Huang ·
Minyi Guo

Published online: 9 August 2011
© Springer Science+Business Media, LLC 2011

Abstract Contexts are often inconsistent in pervasive computing environments, owing to many heterogeneous devices with limited processing capabilities, imperfect measurement techniques, and user movement. A variety of schemes have been proposed to check context inconsistency. However, they implicitly require central control. This requirement inhibits their effectiveness in some pervasive computing environments (e.g., transport systems) where all nodes are resource-constrained and cannot act as a centralized node. To this end, we propose in this paper DCCI—a scheme of Decentralized Checking of Context Inconsistency in pervasive computing environments. DCCI exploits a simple, yet efficient, preference-based locality that denotes nodes requiring that the same context can check the inconsistency on this type of contexts. According to this locality, DCCI constructs a preference-based shortcut structure such that it checks context inconsistency within the shortcut structure. Extensive experiments show that DCCI can accurately and efficiently check context inconsistency in the presence of node churns and heterogeneity.

D. Zhang
School of Computer Science, Nanjing Normal University, Nanjing, China
e-mail: dqzhang@njnu.edu.cn

D. Zhang
Jiangsu Research Center of Information Security & Confidential Engineering, Nanjing, China

M. Chen
School of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea
e-mail: minchen@ieee.org

H. Huang (✉)
College of Computer Science, Chongqing University, Chongqing, China
e-mail: hyhuang@cqu.edu.cn

M. Guo
School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
e-mail: guo-my@cs.sjtu.edu.cn

Keywords Context awareness · Context inconsistency · Pervasive computing

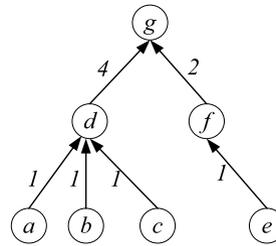
1 Introduction

Pervasive computing is a paradigm shift from traditional desktop computing that constructs an intelligent environment by embedding computation and communication in a way that users accomplish their tasks with minimum intrusiveness [25, 29, 33]. This intelligence is mainly accomplished by context-awareness which is an enabling technology for pervasive computing that assists pervasive applications in automatically adapting to changeable contexts [12]. Contexts refer to the pieces of information that capture the characteristics of pervasive computing environments such as physical contexts and user contexts [13, 32]. They may be acquired by physical sources, i.e., a number of sensing and computational devices, such as hand-held devices, wireless sensors and RFID. They may also be gathered by virtual sources, e.g., inference mechanisms for reasoning users' preferences [35, 36] and virtual sensors for in-network aggregating contexts [17].

Contexts are often inherently imprecise and noisy so that context inconsistency occurs [15, 19, 32]. For instance, a pervasive application detects a conflict in a computation task's context when there are two different locations detected by two sensors for the same user at the same time. 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. Meanwhile, this is also because of the abstraction and complexity of contexts. In asynchronous and heterogeneous pervasive environments, contexts easily become obsolete and dynamically vary with individuals and situations. Detailed reasons for inevitably noisy contexts for pervasive computing can be found in [9, 18, 30]. Note that we assume that every source node can clean and filter its readings, and we only focus on the context inconsistency raised from correct sensor readings.

Consequently, context consistency checking that meets application requirements at runtime to avoid using inconsistent contexts to applications has drawn increasing attention in recent years [6, 30–32]. A variety of schemes have been proposed to check context inconsistency. Most of them require a central node so that the node makes observation globally on a domain and checks context inconsistency based on its knowledge about the relations among domain events. Thus, the fundamental issue is to regulate the access of a number of nodes to the central node in such a way that certain application-dependent performance requirement is satisfied. However, this requirement does not necessarily hold in pervasive computing environments that are saturated with a large number of heterogeneous and equivalent devices (e.g., PDAs and sensors). Due to limited resources and capacity constraints, it is inappropriate to select one from these nodes as a central node. Moreover, this requirement may lead to congestion and unfairness in message-passing. Figure 1 illustrates the shortcomings of central control in pervasive computing environments, where nodes d , g , and f may become the bottlenecks in pervasive applications. All the nodes are in the same environment, and every node collects one type of context and delivers contextual information up to the central node g . The nodes closer to the central node need

Fig. 1 The fairness of central control in detecting context inconsistency. The numbers denote the corresponding link's bandwidth demand



to forward more contextual information than the nodes further away. For example, the nodes d , f , and g are supposed to get 3, 2, and 6 times communication overhead of nodes a , b , c , and e . It is unfair because the nodes d , f , and g will have higher probability of causing congestion than other nodes. In many cases, pervasive computing environments are open, distributed or spread over a large geographical area, and nodes form a dynamic and distributed pervasive network. Nodes frequently join or leave the specific network so that the current central node is not always available and the performance of applications is poor owing to communication cost, delay, resource limitation, and unreliable wireless connection. Thus, the central control scheme for globally capturing contexts and checking context inconsistency is undesirable.

In this paper, we propose DCCI—a scheme for Decentralized Checking of Context Inconsistency in pervasive computing environments which significantly narrows the checking scope of context inconsistency. DCCI leverages a simple, yet efficient, preference-based locality and refers to a supposition that nodes requiring the same context can check the inconsistency of this type of contexts. Then, DCCI builds a simple network overlay. On top of this underlying overlay, DCCI constructs a preference-based shortcut structure such that it checks context inconsistency within the shortcut structure. Thus, once a node in a shortcut list detects a context, it will directly check context inconsistency with the nodes in the same shortcut list. Empirical studies show that DCCI is capable of accurately and efficiently checking context inconsistency in the presence of heterogeneity and mobility. In summary, the contributions of this paper are twofold.

- DCCI is the first scheme in pervasive computing environments for checking context inconsistency in a fully distributed manner. It builds a shortcut structure to significantly reduce the communication overhead and improve the checking accuracy by exploiting a preference-based locality.
- DCCI proposes a powerful primitive for applications using overlay networks that the shortcut structure may be a performance-enhancement, meeting the demands of application customization. For instance, a shortcut structure based on the network latency may reduce hop-by-hop delays in MANETs. In DCCI, we concentrate on a specific shortcut among nodes that imposes various constraints on the same context.

The remainder of this paper is organized as follows. Section 2 briefly overviews the existing schemes on the context consistency checking. Section 3 introduces the elementary concepts on context modeling and context consistency checking. Section 4 discusses the principles and design of the proposed scheme. Section 5 presents our

prototype design and implementation, and reports the experimental results. Section 6 concludes the paper by pointing our future directions.

2 Related work

Context inconsistency detection refers to a process of checking specified constraints on properties of contexts. It is a fundamental issue of context-awareness in cross-discipline pervasive computing, and is receiving increasing attention. In this section, we briefly overview the state-of-the-art research on checking context inconsistency.

A multitude of schemes for checking context inconsistency have been proposed, but they are not fully distributed. In [13], an event detection scheme based on the vector clock for inconsistency checking was proposed, which released an assumption that the contexts being checked belonged to the same snapshot of time. It mapped the inconsistency detection into event detection and modeled event temporal ordering by *happen-before* relationship. It involved a checker process which played a central role in initiating and communication with processes running on nodes. In [30], a context-aware middleware was proposed, which represented contexts and their constraints as tuples. Thus, the middleware designed inconsistency triggers to check context inconsistency by semantic matching among elements of context tuples. The extension of this work, i.e., an incremental scheme of checking context inconsistency based on first-order logic was presented in [31]. In [5, 6], ontologies and assertions were used to model contexts and context inconsistency, respectively. However, ontology-based schemes fall short of removing context noise and dynamically reasoning out hidden contexts. Moreover, they also demand that all ontologies related to a specific domain be pre-defined. Consequently, users have to resort to domain experts, which incurs extra human resource costs.

Note that several schemes for resolving context inconsistency have been initially studied in [2, 3, 20, 23, 32]. However, they implicitly assume that context inconsistency was already detected in some manner. There are also many context-aware schemes for pervasive computing environments have been put forward, consisting of Aura [11], CARISMA [7], CASF [26], Context Toolkit [8], EgoSpaces [16], Gaia [25], and RCSM [34]. They are concerned with either frameworks that support context abstraction and reasoning or context modeling that supports context queries [16, 22], demonstrating that centralized architectures leverage the development and functionality of context-aware applications. In these initiatives, they provided partial support for context management through context-aware frameworks without adequate attention to advanced issues concerning context inconsistency; particularly, inconsistency checking and resolution in a distributed framework were rarely discussed.

Additionally, IBM InfoSphere/InfoStream [14, 21] is a scalable stream processing platform that offers both language and runtime support for sense-and-respond applications in processing data from high rate streams. However, it is limited by two problems. One is that it falls short of checking the inconsistency of contextual data among the most similar node communities. It is a stream processing engine that does not offer preference-based detection of context inconsistency. The other is that Streams

is not cost-effective. It is a commercial product running in clusters and requiring operators to maintain, which incurs high costs.

To summarize, existing schemes of checking context inconsistency are ineffective in asynchronous pervasive computing environments when nodes may form a mobile network. Owing to the frequent churns (node joining and leaving), nodes' heterogeneity and limited capabilities of communication and computation, these schemes cannot always acquire central control. Thus, a problem that the fully distributed manner of checking context inconsistency in the presence of heterogeneity and mobility is raised. To this end, we propose the DCCI scheme that solves the problem. In order to completely and comprehensively understand the proposed scheme, we describe the basic concepts on context modeling and inconsistency checking mechanism before giving a detailed introduction to the DCCI.

3 Preliminary

This section introduces the preliminary knowledge of our research, involving context definition, patterns, and instances.

Contexts are highly abstract, complex, and dependent upon the individuals such that it remains open to define the "context" [1, 4, 28]. The advent of going into the shift from conventional desktop computing to heterogeneous, mobile, and uneven computing environments has gradually stimulated a consensus that contexts comprise the information of users and environments. Herein we share the same viewpoint as [13, 32] and define contexts as follows.

Definition 1 (Context definition) Contexts are the pieces of information that capture the characteristics of pervasive computing environments. To be specific, a context denotes an environmental attribute of a computing entity, which is defined as a context pattern, represented by tuple spaces.

The common contexts related to users consist of user information (e.g., id, location, habits, preferences, emotional state, and bio-physiological conditions), user social environments (co-location of others, social interaction, group dynamics), and the user tasks (e.g., spontaneous activity, engaged tasks, and goals). Likewise, contexts related to physical environments are location (absolute positive, relative position, co-location), infrastructure (e.g., surrounding devices and resources for computing, communication protocols, task performance), and physical conditions (temperature, noise, light, pressure).

Definition 2 (Context pattern) A context pattern (cp) refers to a tuple of fields, i.e., $cp = (owner, attribute, constraint, timestamp)$, where $owner$ is the subject of the context, each $attribute$ represents an environment attribute, each $constraint$ specifies the attribute property of the context, such as temporal and spatial constraints, and $timestamp$ is the context generation time.

The elements $owner$, $attributes$, and $timestamp$ in the context pattern are not allowed to be empty. A context instance $cins$ is generated when the non-empty fields

and some option fields in the context pattern are instantiated. Due to lack of global infrastructures, context instances are not kept in a public space (e.g., a globally-shared tuple space) but their collectors. Thus, all the nodes in pervasive computing environments cannot share the same space of contexts.

Definition 3 (Context instance) A context instance of a context pattern is a tuple $cins = (own:value, attr:value, cons:value, tmp:value)$, where sub , $attr$, $cons$, tmp are the names of the corresponding parts in the context pattern, and $value$ respectively denotes the value of each item.

Suppose that a context about a person location is captured and can be modeled as a context instance $cins = (own:Joanne, location:QP119, cons:uniqueness, tmp:2009-08-08:21:20:30)$. This context instance denotes that Joanne was in room QP119 at 2009-08-08:21:20:30, and the uniqueness constraint indicates that Joanne cannot appear in at two different places at the same time. In this illustrative example, context inconsistency detection is to check the constraints (i.e., uniqueness) on Joanne's location context. If there are more than two locations detected by sensors, the location conflict occurred and pervasive applications may raise a resolution strategy to handle this conflict.

Remark 1 (Context inconsistency detection mechanism) In general, the mechanism of context inconsistency detection is to check whether the certain constraints on properties of context instances or context patterns are satisfied. Once a constraint is violated, the context inconsistency occurs.

Currently, existing schemes for checking context inconsistency are centralized, which may not work in some pervasive applications when nodes form a network with frequent churns. The goal of this paper is checking constraints over various context instances in a fully distributed manner.

4 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 detailed design of DCCI, followed by discussions.

4.1 System 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 $pNeighbor$ nodes, and they form a preference-based list named $pNeighborLst$. In Fig. 2(b), $pNeighbor$ nodes 5 and 8 impose different constraints

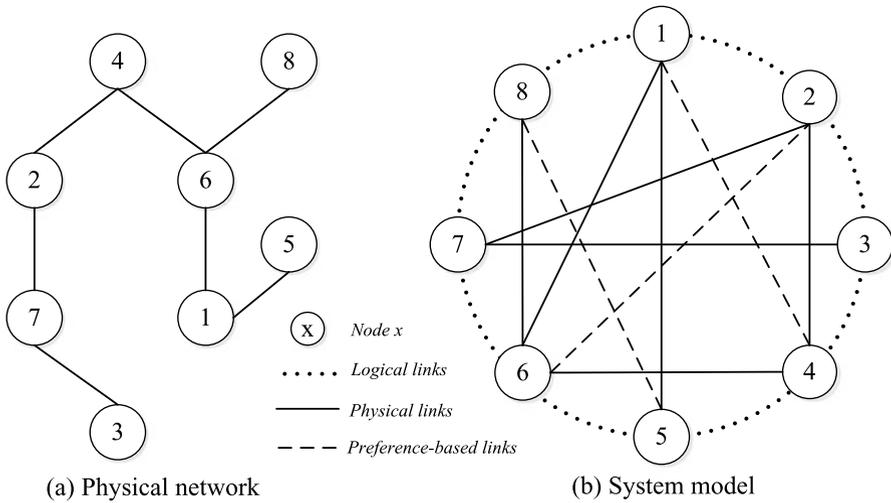


Fig. 2 Illustration of the system model of DCCI, which is constructed by creating shortcuts among nodes that impose diverse constraints on the same context

on the same context and thus they comprise a preference-based list $pNeighborLst$, while $pNeighbor$ nodes 1 and 4 add constraints on a type of contexts and form another $pNeighborLst$. Suppose a node in a $pNeighborLst$ collects a context, it and its neighbors in the $pNeighborLst$ can rapidly check whether context inconsistency does or does not exist 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 contexts being checked 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 cannot 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 $0 \leq v \leq 2$ m/s (the upper bound is set according to the average human walking speed that is about 1.3 m/s) in arbitrary directions to reflect user displacements. At the end of the period t , a node may stay, leave, or move on. 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

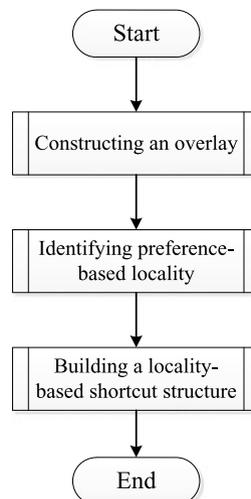
communication can be detected and hidden at the network layer). For example, smart campus is modeled as a distributed pervasive system where students with hand-held devices may frequently switch their scenarios among libraries, classrooms, labs, or mess halls. Every student is modeled as a node, and all hand-held devices are capable of acquiring contexts.

Figure 2 illustrates the system model of DCCI, consisting of three types of links: physical links from the physical network, logical links from the underlying network overlay, and shortcuts from the shortcut structure that is built on top of the network overlay. Physical links mean that nodes can communicate directly, whereas logical links and the shortcut structure represent nodes that can communicate logically. In order to construct the structure, i.e., linking $pNeighbor$ nodes that are in the same $pNeighborLst$, DCCI builds an overlay, identifies nodes that are associated with the $pNeighbor$ relationship, and then discovers shortcuts by the preference-based locality.

4.2 DCCI's Steps

Figure 3 illustrates the DCCI scheme which checks context inconsistency by the 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 Sects. 4.2.1, 4.2.2 and 4.2.3, we will detail these steps. Meanwhile, DCCI provides a maintaining service which is explained in Sect. 4.2.4. It is worth emphasizing that DCCI is compatible with most existing overlays, such as Chord [27], CAN [24], and Gnutella. Pervasive applications are 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.

Fig. 3 The detailed design of DCCI involving three steps: constructing an overlay, identifying preference-based locality, and building a locality-based shortcut structure



4.2.1 Constructing an overlay

This step aims at constructing 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 at all times, thus building a logical overlay.

Concurrent node churns (i.e., nodes joining and leaving 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 hand, 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 its ID is between nodes n_s and n_t . Nodes n , n_t and n_s respectively initiate a *join_check* process. At the beginning, node n sets n_t and n_s as its successor and predecessor. Then, node n checks whether node n_s set it as a successor. If not, node n will notify n_s . Node n does the same process for node n_t . 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.

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 the nodes have to periodically check their neighbors.

4.2.2 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 reduce the cost associated with flooding the network by forwarding only to good relays. However, it is difficult to decide whether an encountered node is a good relay at the moment of encounter.

In order to successfully identify a preference-based locality at low cost, DCCI extends the delegation forwarding protocol by limiting the number of forwarding [10]. The extended protocol helps a node to only forward a message to nodes with quality greater than that of all so far observed nodes for its message. In DCCI, this dramatically reduces the communication overheads.

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:  int iStop, iStart = Timer::getCurrentTime();
12:  iStop = iStart;
13:  while (TRUE) do
14:    iStop = Timer::getCurrentTime();
15:    // default: 3 seconds
16:    while (iStop - iStart  $\geq$  3000) do
17:      if DCCI.getSuccessor( $n_s$ ).unequal( $n$ )
18:        DCCI.notify( $n \rightarrow n_s$ );
19:      else DCCI.setSuccessorState( $n$ ) = true;
20:      if DCCI.getPrecessor( $n_t$ ).unequal( $n$ )
21:        DCCI.notify( $n \rightarrow n_t$ )
22:      else DCCI.setPrecessorState( $n$ ) = true;
23:      iStart = iStop;
24:    end while
25:  end while
26: end procedure

```

4.2.3 Building a locality-based shortcut structure

In this section, DCCI intends to create a shortcut structure by discovering shortcuts among the nodes that impose constrictions on the same context, i.e., linking the $pNeighbor$ nodes in the same $pNeighborLst$ and then checking context inconsistency among the nodes in the $pNeighborLst$.

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 n . Then, node n checks other nodes' context requirements with its requirements by searching over the underlying network. Once a $pNeighbor$ node n' is located, node n will ignore the reply from any other $pNeighbor$ nodes. It will copy the $pNeighborLst$ of node n' , create shortcuts with related $pNeighbor$ nodes, and notify them to add it in their respective $pNeighborLsts$. Thus, nodes in $pNeighborLst$ know their $pNeighbor$ nodes' location, and subsequent context consistency checking go through the $pNeighbor$ nodes with known addresses in the specific $pNeighborLst$. If a node cannot find $pNeighbor$ nodes, it will issue a request to the underlying overlay network. Once a context conflict is detected, the detecting node will

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.setpNeighborLst( $n$ ) =
9:       DCCI.Shortcut.clone( $n'.pNeighborLst()$ );
10:      DCCI.Shortcut.notify( $n \rightarrow$ 
11:      node  $m$  in  $n'.pNeighborLst()$ );
12:      DCCI.Shortcut.addShortcutLst( $n, m$ );

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

```

immediately notify the dependent applications to deal with this conflict. The pseudo-code of constructing a shortcut structure is illustrated as Algorithm 2. Note that lines 5 and 19 are discriminating. The former denotes that nodes do routing in shortcuts that are built by preference-based locality. The latter means that nodes do routing in the underlying network overlay that is constructed by a consistent hashing technique.

In DCCI, the locality-based structure is just a performance-enhancement. If context consistency can be checked in $pNeighbor$ nodes within a specific $pNeighborLst$, 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 so that DCCI cannot detect the location inconsistency. This observation is given as Theorem 1. As a matter of fact, the locality-based structure is easy to customize for application requirements.

Theorem 1 *Shortcuts in DCCI do not affect the correctness of the underlying overlay.*

4.2.4 Maintenance under node churns

In pervasive applications, frequent node churns lead to the change in network topology. Therefore, the shortcut structure must be adapted dynamically. The adaptation

in the underlying overlay is discussed in Sect. 4.2.1, 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' locations 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.

4.3 Discussion

In most cases, DCCI requires $O(\log^2 N)$ messages to reflect the node churns in adapting the underlying overlay and $O(\log N)$ messages to discover shortcuts, where N is the size of the network. It requires $O(1)$ messages to check context consistency by preference-based locality and $O(\log N)$ messages in the worst case. In our implementation, we have two strategies to further improve the performance of DCCI. One is that constrictions are shared in $pNeighbor$ so that, once a context is collected, the consistency may be checked on the spot. The other is that the constrictions in $pNeighbor$ nodes could be reduced in advance. Suppose $pNeighbor$ nodes A and B impose different constrictions on the same context, but the constrictions of B are included in those of A. Thus, once the context satisfies the context requirements of A, context B can be left unchecked.

The locality-based structure can be discovered by several design alternatives such as by searching preference-based locality by flooding across the underlying overlay, or directly searching by various multicast protocols. In this paper, DCCI just studies an elementary protocol regarding the structure for checking context inconsistency. Note that the number of $pNeighbor$ nodes on a specific context is low (less than 10% of all the nodes), i.e., a small number of nodes share the same context with different constraints. The broadcasting technique is preferred when the number of $pNeighbor$ nodes is large. Although DCCI is still compatible, it should limit the scopes of shortcut construction and node selection so that DCCI keeps appropriate shortcut lists in a way that the system performs well in terms of checking accuracy and efficiency.

5 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 the percentage of successfully detecting an inconsistency among all context inconsistencies. In particular, we tried to answer the following questions:

- How large is the success rate of DCCI? Does it work in pervasive computing applications?

- How good is the scalability of the proposed scheme? How many nodes are involved in each context consistency checking? How about the scalability of the proposed scheme when the number of shortcuts added in the system is increased?

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

5.1 Experimental 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 detailed information). We ran experiments on 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 working 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 validity of DCCI in practice. We currently deploy various sensors, e.g., RFID [37] and Bluetooth, into the environment and ask participants to randomly move.

In accordance with the presentation in Sect. 4.1, 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 the nodes. The characteristics of experiment configurations are listed in Table 1.

In our experiments, the overall performance of DCCI is evaluated by the success rate which is defined as the ratio of the successfully checked inconsistencies to the total number of context inconsistencies. The scalability is measured by averaged time and query scope for each context inconsistency checking.

5.2 Success 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 4 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

Table 1 Statistical features 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

Fig. 4 The overall success rate over E500, E2000 and E5000 experiment configurations

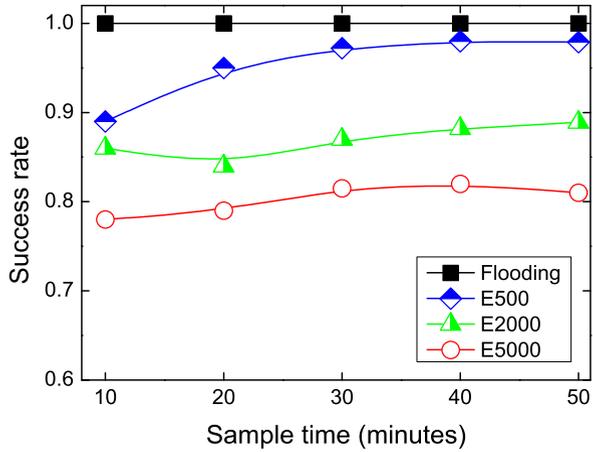
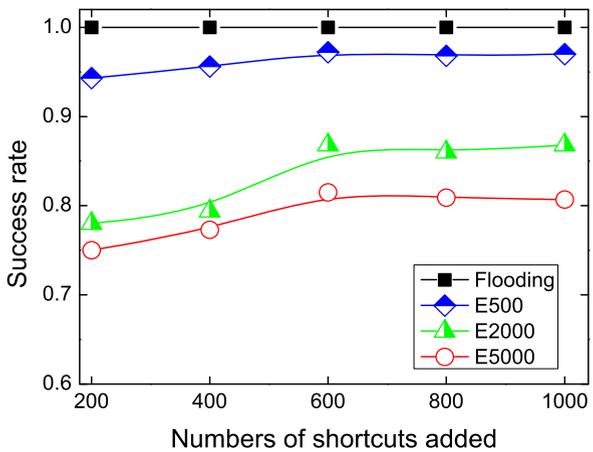


Fig. 5 Number of shortcuts added in context consistency checking over E500, E2000 and E5000 experiment configurations



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.

Figure 5 illustrates how much the preference-based locality affects the averaged success rate, indicating that the more shortcuts are added, the better success rate the 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 higher levels of checking accuracy. After the number of the shortcuts added is about 600, DCCI achieves the best success rate. Then as the number of the shortcuts continues to increase, the success rate diminishes. This is partially because the success rate is also dramatically affected by various environmental factors as it does in Fig. 4. This is also because the sizes

of $pNeighbor$ nodes and $pNeighborLst$ groups lead to a large communication overhead as well as latency, alleviating the performance of preference-based locality.

5.3 Scalability

In this section, we carried out several experiments to evaluate the scalability of the proposed scheme. We first evaluate the overall scalability over the E500, E2000, and E5000 scenarios, and then evaluate how many nodes are involved for each context consistency checking. As discussed above, DCCI aims to use fewer relay nodes when checking context inconsistency by specifying nodes' locations where the context should be transferred to. Compared with the flooding scheme, the scalability of DCCI is chiefly embodied in the time and the number of nodes participating in the process of each context inconsistency checking. Hereby we selected the time and the percentage of nodes involved in the checking process as measurements for scalability.

Figure 6 illustrates the overall scalability of DCCI over various experiment configurations, where the x -axis denotes the experiment configuration, and the y -axis refers to the time spent in experiments. As the experiment configuration increases from E500 to E5000, the time required by the flooding scheme increases exponentially, indicating that it cannot scale well in large-scale pervasive environments. In contrast, the time needed by DCCI increases approximately linearly, meaning that DCCI performs well in terms of scalability. This is because DCCI can accurately locate the nodes where context inconsistency should be checked.

Figure 7 shows how many nodes are involved in the context consistency checking each time. The vertical and horizontal axes are the percentages of nodes participated and experiment configurations, respectively. DCCI significantly reduces the number of participated nodes in consistency checking. For example, in the E5000 configuration, DCCI just required one third of the number of the nodes that were involved in the flooding scheme. The reason for such performance is that DCCI efficiently takes advantage of the preference-based locality so that it accurately locates the desired nodes.

Fig. 6 Scalability over E500, E2000 and E5000 experiment configurations

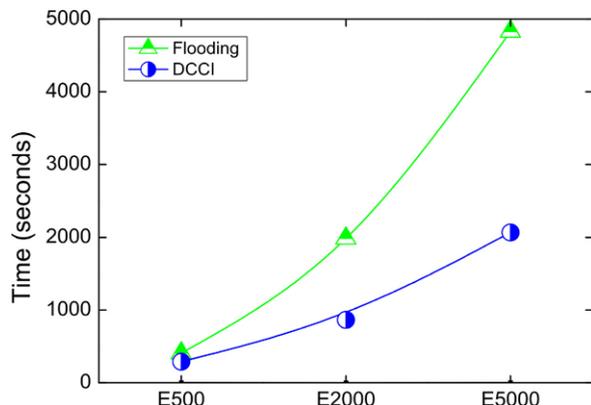
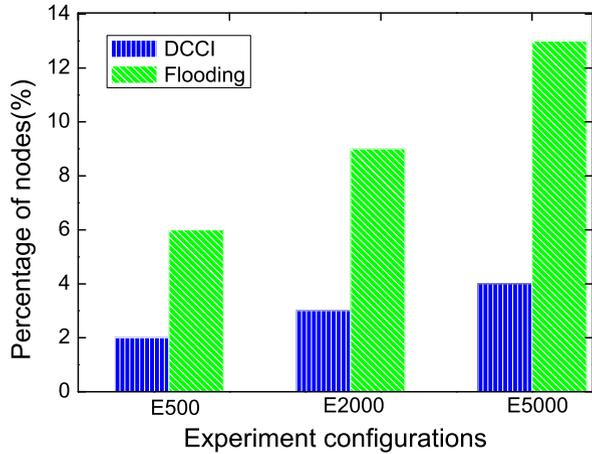


Fig. 7 Percentage of nodes involved in context inconsistency checking over E500, E2000, and E5000 experiment configurations



6 Conclusions

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 of Context Inconsistency—which checks context inconsistency by evaluating the constraints on a 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 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 shortcut-based layer. We will check how many shortcuts should be created so that DCCI can achieve the best performance in terms of accuracy and efficiency for checking context inconsistency. We will incorporate InfoSphere/InfoStream in the data preprocessing module to improve the application scalability and real-time performance. Finally, we will study how to detect and interpret concurrent contexts in asynchronous and dynamic pervasive computing environments.

Acknowledgements We would like to thank Xi Ma for helping us develop and test the codes. We also thank Nicole Kwoh for her technical discussions and proofreading.

This work is supported by the National High Technology Research and Development Program (863 Program) of China (Grant Nos. 2006AA01Z172 and 2008AA01Z106), the National Natural Science Foundation of China (Grant Nos. 60533040, 60725208, 60773089 and 61003247), the Natural Science Foundation of the Higher Education Institutions of Jiangsu Province (No. 11KJB520009) and the Start-up funding at Nanjing Normal University (No. 2011119XGQ0072).

References

1. Baldauf M, Dustdar S, Rosenberg F (2007) A survey on context-aware systems. *Int J Ad Hoc Ubiqu Comput* 2(4):263–277
2. Bikakis A, Antoniou F (2008a) Distributed reasoning with conflicts in a multi-context framework. In: Fox D, Gomes CP (eds) *Proceedings of the 23rd AAAI conference on artificial intelligence (AAAI '08)*, pp 1778–1779
3. Bikakis A, Antoniou G (2008b) Local and distributed defeasible reasoning in multi-context systems. In: *Proceedings of the international RuleML symposium on rule interchange and applications (RuleML '08)*. Springer, Berlin, pp 135–149
4. Bradley NA, Dunlop MD (2009) Toward a multidisciplinary model of context to support context-aware computing. *Hum-Comput Interact* 20(4):403–446
5. Bu Y, Chen S, Tao X, Li J, Lu J (2006a) Context consistency management using ontology based model. In: *International conference on extending database technology*, pp 741–755
6. Bu Y, Gu T, Tao X, Li J, Chen S, Lu J (2006b) Managing quality of context in pervasive computing. In: *Proceedings of the 6th international conference on quality software (QSIC '06)*, pp 193–200
7. Capra L, Emmerich W, Mascolo C (2003) Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Trans Softw Eng* 29(10):929–945
8. Dey AK, Abowd GD, Salber D (2001) A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Int J Hum-Comput Interact* 16(2):97–166
9. Elnahrawy E, Nath B (2003) Cleaning and querying noisy sensors. In: *Proceedings of the 2nd ACM international conference on wireless sensor networks and applications (WSNA '03)*, pp 78–87
10. Erramilli V, Crovella M, Chaintreau A, Diot C (2008) Delegation forwarding. In: *Proceedings of the 9th ACM international symposium on mobile ad hoc networking and computing (MobiHoc '08)*. ACM, New York, pp 251–260
11. Garlan D, Siewiorek DP, Steenkiste P (2002) Project aura: toward distraction-free pervasive computing. *IEEE Pervasive Comput* 1:22–31
12. Harter A, Hopper A, Steggles P, Ward A, Webster P (1999) The anatomy of a context-aware application. In: *Proceedings of the 5th annual ACM/IEEE international conference on mobile computing and networking (MobiCom '99)*, pp 59–68
13. Huang Y, Ma X, Cao J, Tao X, Lu J (2009) Concurrent event detection for asynchronous consistency checking of pervasive context. In: *Proceedings of the 7th annual IEEE international conference on pervasive computing and communications (Percom '09)*, pp 131–139
14. IBM (2010) InfoSphere Streams. <http://www-01.ibm.com/software/data/infosphere/streams/>
15. Jeffery SR, Garofalakis M, Franklin MJ (2006) Adaptive cleaning for RFID data streams. In: *Proceedings of the 32nd international conference on very large data bases (VLDB '06)*, pp 163–174
16. Julien C, Roman GC (2006) Egospaces: facilitating rapid development of context-aware mobile applications. *IEEE Trans Softw Eng* 32(5):281–298
17. Kabadayi S, Julien C, O'Brien W, Stovall D (2007) Virtual sensors: a demonstration. In: *The 26th international conference on computer communications: demonstrations track (Infocom)*, pp 10–12
18. Liu K, Chen L, Liu Y, Li M (2008) Robust and efficient aggregate query processing in wireless sensor networks. *Mob Netw Appl* 13(1–2):212–227
19. Lu H, Chan W, Tse T (2008) Testing pervasive software in the presence of context inconsistency resolution services. In: *Proceedings of the 30th international conference on software engineering (ICSE '08)*, New York, NY, USA, pp 61–70
20. Park I, Lee D, Hyun S (2005) A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. In: *Proceedings of the 29th annual international computer software and applications conference (COMPSAC '05)*, vol 1
21. Pu C, Schwan K, Walpole J (2001) Infosphere project: system support for information flow applications. *SIGMOD Rec* 30:25–34
22. Ranganathan A, Campbell RH (2003) An infrastructure for context-awareness based on first order logic. *Pers Ubiquitous Comput* 7(6):353–364
23. Ranganathan A, Campbell R, Ravi A, Mahajan A (2002) Conchat: a context-aware chat program. *IEEE Pervasive Comput* 1(3):51–57
24. Ratnasamy S, Francis P, Handley M, Karp R, Schenker S (2001) A scalable content-addressable network. In: *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, New York, NY, USA, pp 161–172
25. Román M, Hess C, Cerqueira R, Ranganathan A, Campbell RH, Nahrstedt K (2002) A middleware infrastructure for active spaces. *IEEE Pervasive Comput* 1(4):74–83

26. Satoh I (2009) A context-aware service framework for large-scale ambient computing environments. In: Proceedings of the 2009 international conference on pervasive services (ICPS '09), New York, NY, USA, pp 199–208
27. Stoica I, Morris R, Karger D, Kaashoek MF, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for Internet applications. In: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01), New York, NY, USA, pp 149–160
28. Strang T, Popien C (2004) A context modeling survey. In: Proc of the workshop on advanced context modelling, reasoning and management
29. Weiser M (1991) The computer for the 21st century. *Sci Am* 265:66–75
30. Xu C, Cheung SC (2005) Inconsistency detection and resolution for context-aware middleware support. In: Proceedings of the 10th European software engineering conference held jointly with the 13th ACM SIGSOFT international symposium on foundations of software engineering (SIGSOFT '05), pp 336–345
31. Xu C, Cheung SC, Chan WK (2006) Incremental consistency checking for pervasive context. In: Proceedings of the 28th international conference on software engineering (ICSE '06), pp 292–301
32. Xu C, Cheung SC, Chan WK, Ye C (2008) Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In: Proceedings of the 28th IEEE international conference on distributed computing systems (ICDCS '08), pp 713–721
33. Xue W, Pung H, Palmes PP, Gu T (2008) Schema matching for context-aware computing. In: Proceedings of the 11th international conference on ubiquitous computing (UbiComp '08), pp 292–301
34. Yau SS, Karim F (2004) An adaptive middleware for context-sensitive communications for real-time applications in ubiquitous computing environments. *Real-Time Syst* 26(1):29–61
35. Zhang D, Cao J, Zhou J, Guo M (2009) Extended Dempster–Shafer theory in context reasoning for ubiquitous computing environments. In: Proceedings of the 7th IEEE/IFIP international conference on embedded and ubiquitous computing, pp 205–212
36. Zhang D, Guo M, Zhou J, Kang D, Cao J (2010) Context reasoning using extended evidence theory in pervasive computing environments. *Future Gener Comput Syst* 26(2):207–216
37. Zhang D, Zhou J, Guo M, Cao J, Li T (2011) Tasa: tag-free activity sensing using rfid tag arrays. *IEEE Trans Parallel Distrib Syst* 22:558–570