J. Parallel Distrib. Comput. 74 (2014) 1971–1983

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Detecting stable locality-aware predicates

Min Shen [a], Ajay D. Kshemkalyani [a,*], Ashfaq Khokhar [b]

[a] Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, USA

[b] Department of Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, IL 60607, USA

HIGHLIGHTS

- Algorithms to detect locality-aware predicates in large-scale systems.
- BFS tree topology of height k is used to define locality.
- Can detect stable conjunctive predicates and stable relational predicates.
- Applications to modular robotics and wireless sensor networks.

ARTICLE INFO

Article history:
Received 17 November 2012
Received in revised form
23 July 2013
Accepted 20 September 2013
Available online 29 September 2013

Keywords:
Predicate detection
Locality-aware
Scale-free
Modular robotics
Wireless sensor networks
Distributed computing

ABSTRACT

In a large-scale locality-driven network such as in modular robotics and wireless sensor networks, knowing the state of a local area is sometimes necessary due to either interactions being local and driven by neighborhood proximity or the users being interested in the state of a certain region. We define locality-aware predicates (LAP) that aim at detecting a predicate within a specified area. We model the area of interest as the set of processes that are within a breadth-first search tree (BFST) of height k rooted at the initiator process. Although a locality-aware predicate specifies a predicate only within a local area, observing the area consistently requires considering the entire system in a consistent manner. This raises the challenge of making the complexities of the corresponding predicate detection algorithms scale-free, i.e., independent of the size of the system. Since all existing algorithms for getting a consistent view of the system require either a global snapshot of the entire system or vector clocks of the size of the system, a new solution is needed. We focus on stable LAP, which are those LAP that remain true once they become true. We propose a scale-free algorithm to detect stable LAP within a k-height BFST. Our algorithm can detect both stable conjunctive LAP and stable relational LAP. In the process of designing our algorithm, we also propose the first distributed algorithm for building a BFST within an area of interest in a graph, and the first distributed algorithm for recording a consistent sub-cut within the area of interest. This paper demonstrates that LAPs are a natural fit for detecting distributed properties in large-scale distributed systems, and stable LAPs can be practically detected at low cost.

© 2013 Elsevier Inc. All rights reserved.

1. Introduction

In recent years, distributed systems have found applications in new areas such as modular robotics [1,3,10,12,13,30] and wireless sensor networks (WSNs) [11]. These emerging areas share some common properties such as large scale and dynamic topologies. These properties lead to the need for robust and scalable algorithms to manage, monitor, and reason about the distributed execution in these applications.

A major problem in reasoning about a distributed execution is the detection of distributed properties. The dynamism and nondeterminism of executions present challenges to observing the distributed states of the system. To solve this problem, many distributed predicate detection algorithms have been proposed (see the survey chapter by Kshemkalyani and Singhal [23]). The predicate detection problem is inherently different from the global snapshot problem [4], another well studied problem in distributed systems. A global snapshot gives one of the possible states that could have existed during the period of algorithm execution, while predicate detection verifies the occurrence of the predicate in all possible states that could have existed. Predicates can be either stable or unstable. A stable predicate remains true once it becomes true while an unstable predicate does not satisfy this property. So a snapshot algorithm can detect a stable predicate but not an unstable predicate. In fact, detecting an unstable predicate is an NP-complete problem [23].

Predicate detection has not been studied in large-scale distributed systems. The recent works on snapshots in large-scale systems [15,21,35] take snapshots of the entire system in the

* Corresponding author.
E-mail addresses: mshen6@uic.edu (M. Shen), ajay@uic.edu (A.D. Kshemkalyani), ashfaq@uic.edu (A. Khokhar).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.09.009

context of peer-to-peer (P2P) systems and supercomputer cluster systems. The corresponding algorithms could be used to detect a stable predicate, but they have message complexities of $\Omega(N \log N)$, where N is the number of processes in the entire system.

When the number of processes becomes large and the events are locality driven, users can sometimes be more interested in the state of a local region rather than that of the entire network. The reasons can be that (i) it is too expensive to do a global predicate detection within a large-scale system, or that (ii) properties of local interactions can only be captured by predicates over local regions. Consider the number of patients in a particular emergency room within a WSN monitored hospital environment, or the number of hostile entities in a particular region in a WSN monitored battlefield. In such cases, it makes sense to detect predicates based on a part of the system rather than its entirety. To address such situations, we propose the concept of locality-aware predicates (LAP). Locality-aware predicates are similar to classical predicates. They can also be classified as conjunctive/relational based on the function on the variables involved in the predicate, or stable/unstable based on their detectability. The difference lies in that locality-aware predicates detect a predicate in a sub-network of the system. We call this sub-network an "area of interest". The notion of locality-aware predicates, in the form of "linear distributed predicates", was put forward by De Rosa et al. [11]—however, that paper formalized and gave an algorithm to detect a predicate only within a linear chain or ring topology, which is insufficient to represent a local region. Also, they assumed FIFO channels, which makes it impractical to apply their algorithm in networks such as WSNs where communication channels are not reliable.

We focus on detecting stable locality-aware predicates in this paper, and design an algorithm to detect both stable conjunctive LAP and stable relational LAP. In our proposed solution to detect a stable locality-aware predicate, the user interacts with one process in the area of interest and specifies the range of the area as well as the predicate to be detected. The detection is a 3-stage procedure.

1. The interacted process initiates the construction of an overlay network that corresponds to the local region specified by the user (Algorithm 1 in Section 4). This incurs a one-time cost for establishing the local region.
2. A distributed snapshot within this region is recorded (Algorithm 2 in Section 5) each time the region is to be consistently observed.
3. The recorded snapshot is used for the detection of the predicate (Algorithm 3 in Section 6).

A brief abstract of this work was recently announced [32].

Contributions

1. This paper motivates and proposes the concept of locality-aware predicates in large-scale networks.
2. This paper proposes the first algorithm to detect stable LAP for such networks and assumes non-FIFO channels. The algorithm can detect both stable conjunctive LAP and stable relational LAP. The algorithm is highly efficient and the message count, message space, storage cost, and bandwidth complexities are scale-free, i.e., they are independent of the size of the entire network.
3. To design the above algorithm, the paper also makes the following incidental contributions.
   (a) The paper presents the first distributed algorithm to create a breadth-first search tree (BFST) for a specified region within a network.
   (b) The paper presents the first distributed algorithm to record a consistent snapshot within a specified region of a network.
   The message count, message size, storage cost, and bandwidth complexities of both these algorithms are also scale-free.

Table 1
Summary of notations used.

$(\mathcal{P}, \mathcal{L})$	Undirected network graph defined by the set of processes $\mathcal{P}$ and set of edges $\mathcal{L}$
N	$N = \lvert\mathcal{P}\rvert$, the number of processes in the system
L_{ij}	$L_{ij} \in \mathcal{L}$ is an unidirectional edge from process P_i to P_j
C_{ij}	C_{ij} is a logical communication link (over a path in $\mathcal{L}$) from P_i to P_j
e_i^h	The hth event e executed by process P_i
E_i	$E_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$
$(E, \prec)$	$E = \bigcup_{i,1\le i\le N} e \in E_i$; $\prec$ is the "happens before" or causality relation on E
S_i^t	State of P_i, which is the result of the seq. of events it executed up to the current instant t
m_{ij}	Message sent from P_i to P_j
$send(m_{ij})$	The send event for message m_{ij}
$recv(m_{ij})$	The receive event for message m_{ij}
SC_{ij}^{tu}	The in-transit message set $= \{m_{ij} \mid send(m_{ij}) \in S_i^t \wedge recv(m_{ij}) \notin S_j^u\}$
K	Consistent cut $K \subseteq E$ such that if $e \in K$ then $\forall e' \prec e, e' \in K$
$(\mathcal{P}', \mathcal{L}')$	A sub-graph of $(\mathcal{P}, \mathcal{L})$
K'	Consistent sub-cut of a distributed execution over sub-graph $(\mathcal{P}', \mathcal{L}')$
P_r, P_i	P_r is the initiator of LAP detection; P_i is any process
d	Degree of the graph $(\mathcal{P}, \mathcal{L})$, defined as the maximum of all the processes' degrees
k	The radius of the local region of interest, measured as the number of hops from P_r
n	The number of processes in the local region of interest, and is upper-bounded by d^k
Q	A LAP $Q = (\phi, k, P_r)$, where ϕ is a predicate, k and P_r are defined above

Organization

Section 2 gives the system model and a background on predicate detection as well as snapshot algorithms. Section 3 discusses the challenges in modeling and detecting locality-aware predicates and why a three-stage detection process is necessary. Section 4, which focuses on the first stage, introduces how we model the area of interest and proposes the algorithm to construct an overlay network that represents this area. Section 5, which focuses on the second stage, presents how we modify the existing algorithms to solve the new problem—to construct a snapshot within the area of interest in a large-scale non-FIFO network. Section 6, which focuses on the third stage, presents the algorithm to detect stable conjunctive LAP and stable relational LAP. Section 7 analyzes the complexities of these algorithms in terms of the number of messages, the message sizes, and the storage and bandwidth costs. Section 8 briefly discusses some special cases of detecting stable LAP. Conclusions and future work are discussed in Section 9.

2. Preliminaries

In this section, we introduce the concepts that are related to our work and define the terminology that will be used in later sections. Table 1 summarizes the important notations.

2.1. System model

A distributed system is modeled as an undirected graph $(\mathcal{P}, \mathcal{L})$, where $\mathcal{P}$ is the set of processes and $\mathcal{L}$ is the set of communication links connecting them. Let $N = \lvert\mathcal{P}\rvert$ and $l = \lvert\mathcal{L}\rvert$, and let d denote the degree of the graph, defined as the maximum degree of any process in the graph. The execution of process P_i produces a sequence of events $E_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$, where e_i^h is the hth event at process P_i. An event at a process can be a message send, a message receive, or an internal event. Let $E = \cup_{i,1\le i\le N}\, e \in E_i$ denote the set of events executed in a distributed execution. The causal precedence relation between events induces an irreflexive partial order on E. This relation is defined as Lamport's "happens before"

relation [26], and denoted as $\prec$. An execution of a distributed system is thus denoted by the tuple $(E, \prec)$.

The state of a process P_i at instant t, denoted as S_i^t, is the result of the sequence of all the events executed by P_i up to instant t. For an event e and a process state S_i^t, $e \in S_i^t$ iff e belongs to the sequence of events that have taken process P_i to state S_i^t.

A channel C_{ij} is a logical communication channel from process P_i to P_j. We define C_{ij} to exist if and only if there is a path from P_i to P_j in $\mathcal{L}$. Since the graph is undirected, $L_{ij} \in \mathcal{L} \Rightarrow L_{ji} \in \mathcal{L}$ and C_{ij} exists iff C_{ji} exists. Two processes P_i and P_j are neighbors if $L_{ij} \in \mathcal{L}$. Neighbors can communicate via single-hop communication, whereas non-neighbors communicate via multi-hop communication using logical channels. This distinction is useful for capturing communication cost accurately in large-scale networks such as modular robotics and wireless sensor networks (WSNs). We do not assume FIFO logical channels in this paper even though physical links between neighbors in $\mathcal{L}$ are FIFO. A message sent along channel C_{ij} is denoted as m_{ij} and has two associated events $send(m_{ij})$ and $recv(m_{ij})$ happening at processes P_i and P_j, respectively. The state of a channel SC_{ij}^{tu} depends on the local states S_i^t and S_j^u of the processes on which it is incident and is defined as follows:

$$SC_{ij}^{tu} = transit(S_i^t, S_j^u) = \{m_{ij} \mid send(m_{ij}) \in S_i^t \wedge recv(m_{ij}) \notin S_j^u\}.$$

The superscripts of S_i^t and SC_{ij}^{tu} are not written when they are not important to the context.

A *consistent cut* [4] K is a subset of E such that if $e \in K$ then for $\forall e' \prec e, e' \in K$. We can define a consistent sub-cut K' of a distributed execution as a consistent cut over a sub-graph $(\mathcal{P}', \mathcal{L}')$ of $(\mathcal{P}, \mathcal{L})$. To show that a sub-cut K' is consistent, it is sufficient to show:

1. for all messages whose source and destination are processes within sub-graph $(\mathcal{P}', \mathcal{L}')$, there is no orphan message m_{ij} such that $recv(m_{ij}) \in K' \wedge send(m_{ij}) \notin K'$;
2. if $e_i^h \in K'$, then for $\forall j < h$, $e_i^j \in K'$.

Less formally, a consistent sub-cut K' is a consistent cut with respect to the sub-graph $(\mathcal{P}', \mathcal{L}')$ it resides on.

2.2. Predicate detection

There are several predicate classes studied in the literature (see the survey [23]). We briefly summarize several of the more important classes.

One way of categorizing predicates is based on the function on the variables/states involved in the predicate [6]:

1. A *conjunctive predicate* is a predicate that can be expressed as the conjunction of local variables. For example, $\psi = x_i = 10 \wedge y_j > 20$, where x_i and y_j are variables at processes P_i and P_j, respectively, is a conjunctive predicate. However, determining the number of tokens in a token-passing system is not.
2. A *relational predicate* is a predicate that is expressed as an arbitrary relation on the variables in the system. $\chi = x_i + y_j > 20$ is a relational predicate. Termination, deadlock, and determining the number of tokens are all relational predicates.

Another way to categorize predicates is based on their detectability:

1. A *stable predicate* is a predicate that remains true once it becomes true [4].
2. An *unstable predicate* is a predicate that is not stable and hence may hold only intermittently [6].

A stable/unstable predicate can be either conjunctive or relational. There is literature studying the detection of unstable conjunctive predicates [16,17]. However, due to the exponential complexity, few have studied the unstable relational predicate detection problem.

Stable predicates

Detecting whether a certain stable predicate has become true in an ongoing distributed computation is a fundamental problem for many applications in distributed systems. Examples of stable predicates include termination (the system is in a terminated state with processes in an idle state and no messages in the channels), deadlock (a subset of processes are involved in a circular wait), and garbage collection (an object is a garbage if it has no pointer to it). The ability to detect a stable predicate is vital to application development, including debugging, monitoring, and control. The stable predicate detection problem has been well-studied and many solutions have been proposed for solving the general problem (e.g., [2,4,24,25]) as well as the special cases (e.g., [14,19,27,31]). Stable predicate detection also received attention in recent years (e.g. [8,29]).

2.3. Snapshots in systems with non-FIFO channels

Snapshots have been used to detect stable predicates. A snapshot of a distributed system consists of a consistent collection of local states of processes and a consistent view of the corresponding channel states. The first paper formalizing and solving the global snapshot problem by Chandy and Lamport [4] assumes FIFO channels. Its key idea is as follows. The initiator of a global snapshot recording records its local state and diffuses control messages, called markers, on all channels in the system. A non-initiator process records its local state for the snapshot when it receives the first marker. The messages received in the duration between the local state recording and the arrival of a marker on an incoming channel are recorded as the state of that channel in the global snapshot. The diffusion of the markers contributes to a $O(N^2)$ message count complexity, where N is the total number of processes in the entire system.

Since the Chandy–Lamport algorithm, recording the snapshot in systems with non-FIFO channels has also been studied [7,18,25,28,34]. The details can be referred from a survey [22]. The Lai–Yang algorithm [25] and the Mattern algorithm [28] are *non-freezing* or *non-inhibitory* [7,34], i.e., the algorithms and their control messages do not introduce waits in the underlying computation execution which is being observed and snapshotted/recorded. The algorithm by Helary [18] is inhibitory in contrast. As freezing algorithms are highly undesirable because they interfere with the underlying computation, henceforth, we do not compare our algorithm with that of Helary.

In recent years, the snapshot problem in large-scale distributed systems, such as P2P networks and supercomputer clusters, has also been studied [15,21,35]. The corresponding algorithms could be used to detect stable predicates, but they have message complexities of $\Omega(N \log N)$.

Two techniques from the above papers, the *white/red coloring* by Lai and Yang [25] for recording process states, and the *vector counter* by Mattern [28] for recording channel states, are adapted in our stable LAP detection algorithm.

White/red coloring has these basic rules:

1. A process is initially white and immediately becomes red after it takes a local snapshot.
2. A white (or red) process sends white (or red) colored messages.
3. Upon receiving a red message, a white process takes a local snapshot.

Essentially, a process piggybacks a one bit status information on all outgoing communication messages. This indicates whether or not the process has taken its local snapshot. After every process takes a snapshot of its local state, the set of those local states forms a consistent cut.

The vector counter technique is for recording channel states. This technique requires each process to separately count the number of white messages that it has sent to any other process in a local vector. Every process also counts the number of white messages received from all other processes in one variable. First, a control message circulates through every process, via either a convergecast in a tree topology or a circulation around a ring topology, to calculate the total number of white messages sent to each process. After this, the control message is broadcast to all processes, and each process waits until it has received all in-transit white messages according to the corresponding white message counter in the vector. All the white messages received by P_j from P_i after P_j turns red form the channel state SC_{ij}.

These existing solutions are insufficient for solving the problem of capturing a snapshot under the context of detecting a locality-aware predicate. We expand on this in the next section.

3. Locality-aware predicates

3.1. Motivation

Locality-aware predicates aim at specifying predicates for a local region in a large-scale locality driven network such as modular robotics or WSNs. In such a system, the state of a local region, rather than of the entire system, can be of more interest. This is because: (i) the number of processes in the system is large, thus knowing the state of the entire system can be quite expensive; (ii) the processes' interactions are local, driven by neighborhood proximity; and (iii) the properties of these interactions can only be captured by predicates over local regions.

Consider the following examples. In a token-passing system, the detection of a predicate, Φ = *number of tokens is greater than* 5, defined for the global system might not contain any useful information, since the system contains many processes and the total number of tokens can easily exceed 5. However, if Φ is defined on a local region, then the detection of this predicate provides insight towards this particular region, and thus better captures the interactions within the local region. To detect an explosion event in a WSN deployed field, we need to detect both the temperature and the sound level, as "$temp > 150$ C $\bigwedge$ $sound > 60$ dB". Assuming each sensor can only sense one parameter, this statement is a predicate specified on the local variables of multiple sensors. To make the detection of such a predicate meaningful, the processes whose local variables satisfy this predicate should be close to each other. If the above predicate is detected in sensors which are far apart, then that may not imply that an explosion occurred. In a large-scale locality-driven system, such as WSNs, users are usually interested in such kinds of properties within a certain region. Further examples are the number of patients in a specific area in a WSN monitored hospital environment, and the number of hostile entities in a certain region in a WSN monitored battlefield.

3.2. Detecting locality-aware predicates

When using snapshots to detect predicates, we need to build a consistent cut among the processes over which the predicate is to be detected. For locality-aware predicates, the set of processes over which we detect the predicate is not the entire network. One trivial solution is to take a global snapshot and detect the locality-aware predicate based on a subset of this snapshot. However, the complexity of such a solution is affected by the size of the network. To better solve this problem, we need to design algorithms that are *scale-free*, meaning that the size of the entire system does not affect the complexity of the algorithms. For this purpose, we need to take the snapshot only within the area of interest.

Table 2
Comparison of features of algorithms for detecting stable predicates in large-scale distributed systems.

Feature	Non-freezing	Non-FIFO channel	Locality awareness	Scale-free
Chandy–Lamport [4]	√	×	No	×
Modified C–L (Section 3.2)	√	×	Spanning tree	×
Mattern's non-FIFO Snapshot [28]	√	√	No	×
LDP-Basic [11]	√	×	Linear chain	√
LDP-Snapshot [11]	×	×	Linear chain	√
LAP Algorithms	√	√	BFST or any tree	√

One seemingly possible solution to design a scale-free algorithm for recording a snapshot within the area of interest would be to run the Chandy–Lamport snapshot algorithm with hop count, that is, to count the number of hops the marker message has traversed from the initiator and stop sending markers once the hop count reaches a certain value. We term this algorithm as the modified Chandy–Lamport (modified C–L) algorithm. It has three drawbacks.

1. The overlay network that this algorithm constructs is a spanning tree. If we want to cover all processes within a certain distance from the initiator, a spanning tree is not sufficient since the spanning tree does not find the shortest paths.
2. Using the modified C–L snapshot algorithm will require the communication channels to be FIFO. This is not practical in networks such as WSNs.
3. Most importantly, the modified C–L algorithm can go wrong with the recorded process states and the recorded channel states within the area of interest. This is because it is unable to track messages that transitively traverse outside the area of interest and potentially reenter the area. Fixing this problem requires taking a system-wide global snapshot, which will make the solution non-scale-free.

In essence, the modified C–L algorithm cannot construct a consistent sub-cut over the area of interest in a scale-free manner.

Besides the Chandy–Lamport snapshot algorithm, all the existing non-FIFO snapshot algorithms [15,18,21,25,28,35] cannot be directly applied to solve the problem in a non-FIFO network. There are two reasons. First, most of the existing algorithms rely on a spanning tree overlay network [18,25,28] or an even more rigid overlay network such as a hypercube [15,21,35]. Second, when capturing the process states and the channel states, the existing algorithms are designed for the entire network and will cause the complexity to be non-scale-free.

So, a new solution is needed. It needs to be scale-free, capable of constructing an overlay network that represents the area of interest, and effectively and efficiently takes a snapshot within the area of interest. In addition, it detects a predicate within the area of interest. We design such a solution as a three-stage procedure, as outlined in Section 1.

We now compare features of potential solutions to solve the problem of detecting a stable LAP in Table 2. We compare the Chandy–Lamport snapshot algorithm [4], the modified C–L algorithm, and Mattern's non-FIFO snapshot algorithm [28]. We also compare with the two algorithms, *LDP-Basic* and *LDP-Snapshot* [11], that can detect a locally distributed predicate (LDP) within a *linear topology* only. Both these algorithms work only if the communication is single-hop to direct physical neighbors, and the underlying channels are FIFO. *LDP-Basic* can only detect a stable predicate that does not depend on channel states, while *LDP-Snapshot* uses freezing to detect a stable predicate that may depend on channel states. Lastly, we compare with our proposed solution, which we term as LAP algorithms. The Lai–Yang algorithm [25]

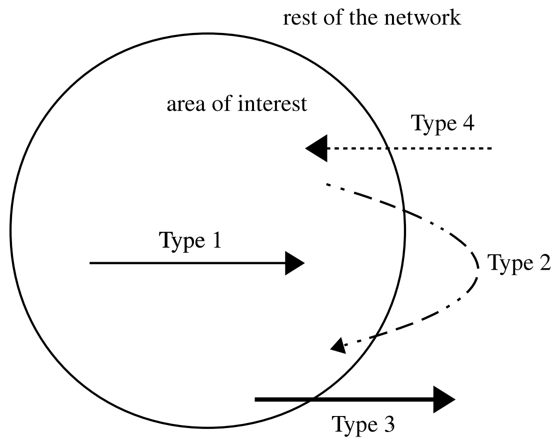

Fig. 1. Four types of messages.

requires unbounded memory to track all past messages, whereas the Helary algorithm [18] is freezing; both these algorithms have highly undesirable features and hence are not considered.

Some classification to design our solutions (LAP algorithms) is introduced next. When taking a snapshot, we need to ensure there is no orphan message [23] present in the snapshot. For the processes within the area of interest, messages can be categorized into 4 types, as illustrated in Fig. 1.

1. Messages transmitted entirely within the area of interest,
2. Messages whose source and destination are within the area of interest, but some of the transmitting intermediate processes are outside the area,
3. Messages sent from within the area of interest to some processes outside it,
4. Messages sent from outside the area of interest to some processes within it.

Since our goal is to take a snapshot for processes within the area of interest, type 3 and 4 messages can be ignored when checking for orphan messages. This is because whether such messages are orphan messages depends on the states of processes outside the local region. So, theoretically we need to check that there are no orphan messages only among type 1 and 2 messages. To classify a message as one of the four types, the algorithm needs to know the source and destination processes, and whether they belong to the area of interest. Although a scale-free solution of space complexity $O(d)$ might seem possible by simply tracking whether each neighbor of a process in the area of interest is also in the area of interest, this will not work because channel states may not be captured correctly. This is because each pair of processes P_i and P_j communicate over a logical channel C_{ij}. Even though P_i and P_j may be in the area of interest, they may not be neighbors in $(\mathcal{P}, \mathcal{L})$; and therefore it is necessary to know whether each other process is in the area of interest. Furthermore, a message sent along C_{ij} may traverse outside the area of interest over physical links and hence it is necessary to record the states of logical links rather than physical links in the area of interest.

4. Modeling area of interest

The key aspect of specifying a locality-aware predicate is to specify the area of interest. So the first stage of the solution is to construct a topology that can represent the area of interest. We want to detect the predicate in an area centered at the process P_r the user interacts with and the "radius" of the area is k, meaning that processes in the area are within distance k in terms of the number of edges from P_r. This circular region is a natural model for the area of interest, particularly in WSNs and modular robotics applications because it captures geographical proximity. To achieve this, we need a topology that covers all the processes in such an area; a simple spanning tree will not suffice because it may not include all the processes within k hops from P_r; see Fig. 2 for an illustration of this concept. For this purpose, we use a local breadth-first search tree (BFST) as the topology to model the local region. The local BFST is rooted at process P_r with height k.

A distributed algorithm to construct a BFST was given by Chandy and Misra [5]. However the BFST is constructed for the entire network and their algorithm has an $O(N^2 d)$ message count complexity. To construct a local BFST, we face the challenge of determining when the algorithm terminates. Trivial solutions such as the asynchronous Bellman–Ford algorithm [23] do not have any mechanism to determine when to terminate. Although the Chandy–Misra algorithm [5] proposed a way to determine termination by counting the number of acknowledgments, we cannot directly adapt that algorithm by using a hop restriction. This is because a process temporarily k hops away from P_r may later discover a shorter path. Thus, we need to design a solution that can correctly and efficiently determine when the algorithm terminates. Compared with the Chandy–Misra algorithm [5], our algorithm also generates fewer acknowledgment messages. Besides that, our algorithm is capable of determining the list of children for each process in the local BFST when the algorithm terminates. This is important for the later stages in our stable LAP detection solution.

We use two types of control messages in the algorithm *LocalBFST* (P_r, k), which is listed as Algorithm 1.

1. A message *length* (s, P_j, k) received at P_i indicates that there is a path of length s from P_r to P_i with P_j being the predecessor of P_i. The distance limitation k is also contained in this message.
2. A message *ack* (*positive*/*negative*, s, *pids*) acknowledges a *length* message sent from P_j to P_i after a certain condition is met. An *ack* can be either positive or negative, and also carries the length parameter s of the corresponding *length* message. The parameter *pids* is a set that contains the process identifiers of some of the nodes in the sub-tree traversed. This parameter is non-empty only on positive *ack*s.

Each process P_i also maintains several local variables.

1. *dist*: the length of the shortest path from P_r to P_i known so far. P_r initializes *dist* to 0, other processes initialize *dist* to ∞.
2. *pred*: the predecessor on the shortest path from P_r to P_i known so far, and is initially undefined. The message *length* (*dist*, *pred*, k) is received from *pred*.
3. *num*: the number of unacknowledged *length* messages, initialized to 0.
4. *child_list*: a list of processes that become a child process of P_i in the tree topology, initialized to the empty list.
5. T: a set of some of the process identifiers in the sub-tree, initialized to Ø.

Correctness

Observation 1. *Once a process receives a length message, the process will always have some parent pred and will always be a part of the local BFST.*

Observation 2. *For a process P_i in the local BFST, its neighbors that are also in the local BFST are exactly those processes P_j to which P_i sent a length message or from which P_i received a length message.*

Observation 3. *The variable dist is a strictly decreasing function and can change at most k times.*

Theorem 1. *Algorithm* 1 *identifies all the processes in the local BFST, defined as a BFST rooted at P_r with height k.*

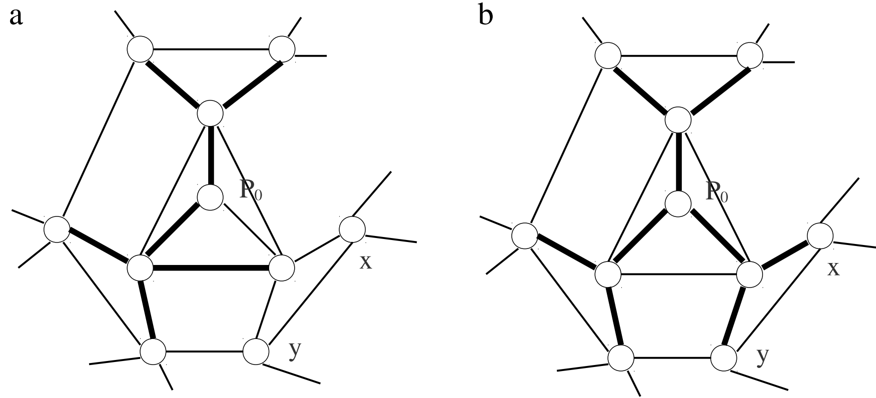

Fig. 2. Illustration of covering an area centered at P_0 with radius 2. Bold edges are tree edges. (a) A spanning tree might exclude some processes such as x and y because it does not find shortest paths. (b) A local BFST includes all the processes.

Algorithm 1 Local BFST (P_r, k)

i. P_r initiates the construction of local BFST:
1. $dist = 0$, *pred* undefined, *child_list* empty, $T = \{i\}$;
2. send $(1, P_r, k)$ to all neighbors;
3. $num = \#$ of neighbors;

ii. P_i receives $length(s, P_j, k)$ from P_j:
4. **if** $(s < dist)$ **then**
5. **if** $dist = \infty$ **then**
6. $T = T \bigcup \{i\}$;
7. **if** $(num > 0)$ **then**
8. send $ack(negative, dist, \emptyset)$ to *pred*;
9. $pred = P_j$; $dist = s$; $child_list = \langle\rangle$; $num = 0$;
10. **if** $(s + 1 \leq k)$ **then**
11. send a message $length(s + 1, P_i, k)$ to every neighbor except P_j;
12. $num = \#$ of neighbors -1;
13. **if** $(num == 0)$ **then**
14. send $ack(positive, dist, T)$ to *pred*;
15. $T = \emptyset$
16. **else**
17. send $ack(negative, s, \emptyset)$ to P_j;
18. remove P_j from *child_list*;

iii. P_i receives $ack(*, s, T')$ from P_j:
19. $T = T \bigcup T'$
20. **if** $(s \neq dist + 1)$ **then**
21. discard the *ack* message
22. **else**
23. $num = num - 1$;
24. **if** (*ack* is positive) **then**
25. add P_j to *child_list*;
26. **if** $(num == 0)$ **then**
27. **if** $(pred \neq P_i)$ **then** // non-root
28. send $ack(positive, dist, T)$ to *pred*;
29. $T = \emptyset$
30. **else** // for P_r
31. broadcast $terminate(T)$ on local BFST;
32. terminate the algorithm;

iv. P_i receives $terminate(X)$ from P_j:
33. X identifies the process set in the area of interest;
34. propagate the $terminate(X)$ on local BFST;
35. terminate the algorithm.

Proof. Define $min_dist(P_i)$ as the length of the shortest path from P_r to P_i. We say that a process gets engaged by a message $length(x, *, *)$ if $x < dist$ at the process. By Observation 1, the process will be part of the local BFST. We prove the theorem by induction on $min_dist(P_i)$, using the hypothesis, "If $min_dist(P_i) = x \leq k$, then P_i is included in the local BFST".

$min_dist(P_i) = 1$: P_r sends $length(1, P_r, k)$ to all its neighbors. The last engagement of any process P_i having $min_dist(P_i) = 1$ is by the $length(1, P_r, k)$ it receives. By Observation 2, P_i is included in the local BFST.

$min_dist(P_i) = x$ $(x \leq k - 1)$: Assume the induction hypothesis is true.

$min_dist(P_i) = x + 1$ $(x \leq k - 1)$: By the induction hypothesis, each process P_j such that $min_dist(P_j) = x$ gets last engaged by a message $length(x, *, k)$, and by line (11), sends $length(x+1, P_j, k)$ to all its neighbors except *pred*, where $min_dist(pred) = x - 1$. Thus, any process P_i such that $min_dist(P_i) = x + 1 \leq k$ will receive at least one $length(x + 1, *, k)$ message, and get last engaged by the first such message. By Observation 2, P_i is included in the local BFST.

Further, if $min_dist(P_i) = k$, by line (11), P_i will not send any $length(k + 1, P_i, k)$ messages, and no process P_j having $min_dist(P_j) > k$ will ever be engaged and will not be identified as part of the local BFST. □

Algorithm 1 is guaranteed to terminate correctly. When process P_i sends the *length* messages to its neighbors, the counter *num* is set to the number of *length* messages sent (line 12). Whenever an *ack* is received, either positive or negative, the counter decreases by 1 (line 23). Furthermore, each time P_i discovers a shorter path, it will reset its counter to 0 (line 9). We also make sure that P_i will only decrease its counter when the distance marked in the *ack* message received matches P_i's current *dist* (lines 20–21). By performing these operations, P_i can know whether all *length* messages corresponding to its current *dist* have been acknowledged. This is guaranteed to happen because for each *length* message generated, exactly one *ack* message will be sent back. When this happens, all the processes in the temporary sub-tree rooted at P_i (this sub-tree might still change if any process discovers a shorter path via some process outside this sub-tree) also have their *num* being 0. When P_r's *num* becomes 0, the counter *num* at all processes in the local BFST must have already turned 0. This ensures that all processes in the local BFST have been discovered and every process in the local BFST has discovered the shortest path, because otherwise there will be at least one process whose *num* is not 0.

Upon termination at the root P_r, all processes in the area centered at P_r with radius k form a local BFST, and are also identified at P_r. The identifiers of all the processes get collected in the T parameter at the root. Consider any P_i that has received some *length*

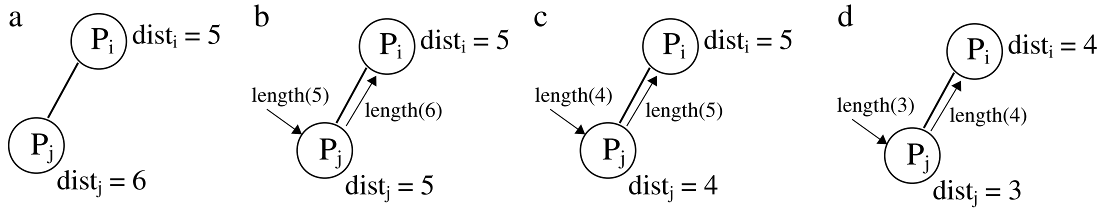

Fig. 3. Illustration of dynamic changes to *child_list*. (a) Initially P_j is P_i's child, with P_i's *dist* being 5 and P_j's *dist* being 6. (b) P_j discovers a shorter path with *dist* being 5 and sends a *length* message to P_i. (c) P_j discovers a shorter path with *dist* being 4 and sends a *length* message to P_i. (d) P_j discovers a shorter path with *dist* being 3 and sends a *length* message to P_i. In (b) and (c), P_i can discover P_j is no longer its child and can safely remove P_j from its *child_list* in line (18). In (d), P_i becomes P_j's child and resets its *child_list* in line (9).

message. P_i sends $i \in T$ to the first parent P_j to which it sends a positive *ack*. (P_j has a smaller *dist* value.) P_j will add i to its T variable (line (19)). Observe from lines (13–14) and (26–28) and Observation 3 that P_j has not yet sent any positive *ack*, or if it has, it will still send one more to a new parent. P_j's T variable containing i is now included in the first positive *ack* it sends to some parent, say P_k. (P_k has an even smaller *dist* value.) Inductively, within k hops, P_i's identifier i reaches P_r on some positive *ack*, after which P_r terminates. Thus, every process that has received a *length* message is included in T at P_r.

Each process in this local BFST also knows its parent (*pred*) and children (*child_list*) upon termination. Each process' parent is correctly identified because *pred* is always set to the predecessor on the shortest path known so far. For the list of children, each time a positive *ack* is received at P_i from P_j, P_j has already set its *pred* to P_i. Thus, P_i adds P_j to its *child_list*. However, it is possible that P_j later discovers a shorter path and sets its *pred* to a different process. Thus P_i needs to remove P_j from its *child_list*. This is achieved in our algorithm. Notice that when P_j discovers a shorter path, it will send *length* messages to all its neighbors, except the predecessor on the shorter path; thus, a *length* message gets sent to P_i. When this happens, only one of 3 situations could occur, as illustrated in Fig. 3(b–d). In each case, P_i can discover that it is no longer P_j's parent and can safely remove P_j from its *child_list*: this happens in line (18) for cases (b,c) and in line (9) for case (d).

This guarantees that for any process P_i, its *child_list* contains all and only all the processes that becomes P_i's children.

When the root terminates, it needs to broadcast a *terminate* message in the local BFST, in order for other processes in the local BFST to learn of the termination. This is because they would not otherwise know if a shorter path than that per their current knowledge is still being searched. The non-root processes terminate when they receive the *terminate* broadcast from their parent, *pred*. All the local BFST processes also learn of which other processes are in the local BFST when they receive the T parameter on the *terminate* message.

We denote the total number of processes in the entire system as N and the number of processes in the area of interest as n. The complexity of Algorithm 1 is scale-free (to be shown in Section 7), meaning that its complexity is affected only by n, k, and d, but not by N. This is a feature shared by all the algorithms introduced in this paper. Being scale-free is important to locality-aware predicate detection because a locality-aware predicate models a property within a local region. Having an algorithm with complexity relative to N will make it non-scalable and thus not applicable in large-scale distributed systems. We will give the detailed complexity analysis in Section 7.

With Algorithm 1, we ensure that in the first stage we can dynamically construct an overlay network covering all the processes in the area of interest with a relatively low cost. Note that Algorithm 1 is not robust to churn because of the intricate interactions among the various *length* and *ack* messages.

5. Consistent sub-cut construction

Now, we assume that Algorithm 1 has already run and a local BFST rooted at P_r is constructed. In the second stage, a snapshot within the area of interest is taken. We base our algorithm on top of the *white/red coloring* and *vector counter* techniques discussed in Section 2.3. For capturing process states, the *white/red coloring* technique is sufficient to identify pre-recording and post-recording messages in non-FIFO systems. Also, it does not incur any extra overhead besides associating a one bit data with the messages. However, for capturing channel states, the *vector counter* technique does not solve the problem in a scale-free manner. This technique is designed for capturing channel states while taking a global snapshot. It also ensures that all the in-transit messages while taking the snapshot get delivered to the destination when the algorithm terminates. Directly applying this technique will incur an $O(N)$ storage cost and bandwidth cost, thus causing the solution to be non-scale-free. Instead of maintaining a size N vector to count white messages sent to/received from every process in the system, each process P_i in the area of interest only maintains a size $n = |T|$ vector to count the white messages traversing on logical channels to processes within the area of interest. Recall that T is the set of processes in the local BFST, as computed by Algorithm 1. We assume the n processes in T are $P_{l_1}, P_{l_2}, \ldots, P_{l_n}$ and P_r associates this mapping, which maps the ID i of process $P_i = P_{l_a}$ in the local BFST to a virtual ID a in the range $[1, n]$, with the *terminate* message in Algorithm 1. Thus every process has a unique position in a size-n vector. This is important for the local variables introduced later.

As we need a scale-free algorithm, no process can start coloring the messages white at system initialization time because it does not know the area of interest, and hence would have to track the messages sent to all N processes. As the local BFST is formed on-the-fly, we are faced with the challenge of identifying (i) when to begin coloring the messages as white, (ii) when to begin counting the white messages sent to each other process in the BFST, and (iii) when to start counting the white messages received. These operations are essential to ensure that the recorded channel states are complete. We claim that no coordination is needed among the processes in the local BFST to begin these operations.

Observation 4. *A process in the BFST can begin coloring messages sent to processes in the BFST as white and counting the white messages sent to (and received from) others in the BFST, at any time before recording its local snapshot. The count of incoming white messages should begin no later than receiving the first white message.*

This follows from the fact that each logical channel is independent and the message count is per logical channel. The above observation is implemented in the one-time pre-processing for the algorithm.

So, for our algorithm, each process P_{l_i} maintains the following local variables:

1. $color_{l_i}$ records the color of P_{l_i} as either white or red; initialized to white.

2. $wmsg_sent_{l_i}[1 \ldots n]$ is a vector of size n. $wmsg_sent_{l_i}[j]$ counts the total number of white messages sent to process P_{l_j} in the local BFST.
3. $wmsg_to_recv_{l_i}$ is an integer variable which accumulates the count about the total number of white messages it should receive (from processes that are also in the local BFST) in order to complete the recording of channel states for the consistent sub-cut.
4. $wmsg_recvd_{l_i}$ is an integer variable which counts the number of white messages received (from processes that are also in the local BFST).

There are three types of control messages in the algorithm:

1. An *INIT* message gets broadcast within the area of interest via edges of the local BFST constructed in the first stage. It initiates the algorithm and serves as a red communication message in case some process in the local region has not received any communication messages from other processes in the same local region.
2. A *Cvg_Acc_White* message convergecasts the *wmsg_sent* vector.
3. A *Bcast_Acc_White* message broadcasts the *wmsg_sent* vectors accumulated at the root.

In addition to these control messages, each process will also receive white or red communication messages which they also need to handle accordingly.

Our algorithm is listed in Algorithm 2. The one-time pre-processing after the local BFST is constructed involves the following action.

- As part of the initialization, recall that in Algorithm 1, the root broadcasts the $terminate(T)$ on the local BFST to inform the processes of the IDs of the processes in the local BFST. As each process gets engaged by the broadcast, it now initializes the size-n vector $wmsg_sent[1 \ldots n]$, starts coloring messages white, and counting the number of such messages sent to each other process in the local BFST, and received from processes in the local BFST. (If a white message is received before the $terminate(T)$ broadcast is received, the $wmsg_recvd_{l_j}$ variable is updated right away.)

For each snapshot to be collected after the pre-processing step, the algorithm is executed in five phases:

1. The root of the local BFST initiates a one-to-all broadcast of an *INIT* control message along the BFST edges to inform all processes in the tree about the commencement of taking the snapshot.
2. The number of white messages sent along each logical channel whose both end-points are incident on processes within the local BFST is determined in this phase along with recording the local snapshot. First, upon receiving the *INIT* control message or a red computation message, a white process turns red and records its local state. A process might have already turned red before receiving the *INIT* message; in this case, it simply ignores the *INIT* message. Second, when a leaf node P_{l_i} turns red, it initiates a convergecast to component-wise accumulate the $wmsg_sent_{l_i}[1 \ldots n]$ vector in the local BFST.
3. After executing the second phase in which $Cvg_Acc_White(W)$ accumulates the $wmsg_sent_{l_i}$ vectors of all nodes in the local BFST at the root, the root initiates a $Bcast_Acc_White(W)$ to inform each process in the local BFST of the number of white messages sent to it up to the snapshot recording.
4. As each process gets engaged by the *Bcast_Acc_White*, it saves in $wmsg_to_recv_{l_i} = W[i]$ the number of white messages it should receive in order to complete channel state recording.
5. The channel states are recorded in this phase. When a red process P_{l_i} receives a white or an uncolored message from P_{l_j} (which is also in the local BFST), P_{l_i} adds such a message to the channel state $SC_{l_jl_i}$ and increments $wmsg_recvd_{l_i}$ if the message was white. Once P_{l_i} determines that $wmsg_to_recv_{l_i} = wmsg_recvd_{l_i}$, it terminates the algorithm locally. It is now ready to participate in the LAP evaluation (Algorithm 3).

Algorithm 2 Construction of Consistent Sub-cut(P_r, k, ϕ)

Initialization: P_{l_i} initializes the algorithm at the end of Algorithm 1:
1. P_r: On broadcasting $terminate(T)$, initialize $wmsg_sent[1 \ldots n]$, start coloring messages white and start counting white messages sent to processes in T. Also operate $wmsg_recvd$.
2. $P_{l_i} \in$ local BFST: On receiving $terminate(X)$, initialize $wmsg_sent[1 \ldots n]$, start coloring messages white and start counting white messages sent to processes in T. Also operate $wmsg_recvd$.

i. P_r starts collecting the snapshot:
1. send an $INIT(\phi)$ message to all processes connected by local BFST edges and to P_r itself;

ii. White process P_{l_i} receives an *INIT* or a red communication message from P_{l_j}:
2. $color_{l_i} =$ red;
3. **if** ($P_{l_i} \neq P_{l_j}$) **then**
4. send an *INIT* message to all children in local BFST;
5. record local state pertinent to ϕ;
6. **if** (P_{l_i} is a leaf node in local BFST) **then**
7. initiate convergecast $Cvg_Acc_White(wmsg_sent[1 \ldots n])$;

iii. P_{l_i} receives $Cvg_Acc_White(W[1 \ldots n])$ from P_{l_j}:
8. participate in convergecast by accumulating the W vectors from all children in local BFST and its own $wmsg_sent$ vector recorded in the local snapshot;
9. **if** $P_{l_i} \neq P_r$ **then** // non-root
10. send $Cvg_Acc_White(W)$ to parent;
11. **else**
12. initiate broadcast $Bcast_Acc_White(W)$;

iv. P_{l_i} receives an $Bcast_Acc_White(W)$ message from P_{l_j}:
13. $wmsg_to_recv_{l_i} = W[i]$;
14. **if** P_{l_i} is not a leaf node **then**
15. propagate $Bcast_Acc_White(W)$ to child nodes;

v. Red process P_{l_i} receives a white or uncolored message *msg* from P_{l_j}:
16. **if** (P_{l_j} is in local BFST) **then**
17. record *msg* as part of channel state $SC_{l_jl_i}$;
18. **if** (*msg* is white) **then**
19. $wmsg_recvd_{l_i}$++;
20. **if** ($wmsg_to_recv_{l_i} = wmsg_recvd_{l_i}$) **then**
21. terminate the algorithm locally.

Correctness

To prove that Algorithm 2 is correct, we need to show that the process states do not contain an orphan message and that the channel states are complete.

- Since a white process records its local state upon receiving the first red communication message or the *INIT* message, there will be only white communication messages in its recorded local state. Also, since a white process turns red after recording its local state, all the *send* events of white messages would have

been recorded. Thus, the process states do not contain orphan messages.

- Since the algorithm does not terminate locally at a red process until it has received the correct number of white messages on each incoming channel from any other process also in the local BFST (line 20), this guarantees that the channel states are complete.
  In our algorithm, as the messages are colored white and counted only after the local BFST is formed, no additional overhead is required before the detection of a stable LAP. However, the first snapshot recorded by our algorithm may record some incomplete channel states. This is because some (uncolored) message generated before the pre-processing step may get delivered to the destination after the snapshot and channel state recording is completed at the destination. Such a potentially incomplete channel state recording can be avoided by introducing a small delay between the initialization step and step (i). This is because (even with non-FIFO channels,) messages will eventually be delivered. In wireless transmission protocols such as IEEE 802.11 [20], although the value of the ACK timeout is not defined in the specification, the general setting is SIFS + ACK + Round Trip Propagation Delay [9], which is usually tens of microseconds. Repeated invocations of Algorithm 2 are often needed to test for a stable LAP, and are sequential and spaced apart. Hence, for the second and subsequent invocations, the possibility of an incompletely recorded channel state becomes zero very quickly.

Observe that the definition of a consistent sub-cut (Section 2.1) is not concerned with transitive inconsistencies caused by Type-2 messages of Fig. 1. However, observe that we can easily modify Algorithm 2 to prevent inconsistencies caused by Type-2 messages as follows: instead of coloring with two colors, sub-cut snapshot sequence numbers are needed. The message coloring rule is modified to use sequence numbers. All the processes in the system follow this rule but only the processes in the local BFST execute Algorithm 2.

After running Algorithm 2, the snapshot within the area of interest is recorded. This provides the foundation for detecting locality-aware predicates.

6. Detecting locality-aware predicates

The third stage of detecting locality-aware predicates (LAP) is to actually detect the predicate based on the recorded distributed snapshot. Section 6.1 formally defines a LAP and gives examples. Section 6.2 presents the algorithm for detecting stable LAP, and discusses how it can be adapted for conjunctive LAP and for relational LAP.

6.1. Formal definition

Definition 1. A *LAP* $Q = (\phi, k, P_r)$ is a predicate ϕ over the states of processes within the local BFST rooted at P_r with height k.

- If ϕ is conjunctive, then LAP Q is conjunctive.
- if ϕ is relational, then LAP Q is relational.

Examples of conjunctive LAP

- $Q_1 = (\phi, 3, P_r)$, where $\phi = \wedge flag_{P_i}$, for P_i in a height-3 local BFST rooted at P_r.
  Q_1 is true if each process in P_r's local region with radius 3 has set its flag.
- $Q_2 = (\phi, 5, P_r)$, where $\phi = \wedge terminated_{P_i}$, for P_i in a height-5 local BFST rooted at P_r.
  Q_2 is true if each process in P_r's local region with radius 5 has terminated.
- $Q_3 = (\phi, 5, P_r)$, where $\phi = \wedge temp_{P_i} > 50$, for P_i in a height-5 local BFST rooted at P_r.
  Q_3 is true if each sensor in P_r's local region with radius 5 has a temperature reading greater than 50° F in a WSN monitored field.

Examples of relational LAP

- $Q_4 = (\phi, 5, P_r)$, where $\phi = \sum_{P_i \in local\ BFST} token_{P_i} \geq 3$.
  Q_4 is true if there are at least 3 tokens among processes within an area of radius 5 from P_r.
- $Q_5 = (\phi, 6, P_r) = average_of_temp_{P_i} \geq 50$, for P_i in the local BFST rooted at P_r.
  Q_5 is true if (in a WSN monitored field) the average temperature sensed within an area of radius 6 from P_r is larger than 50° F.

6.2. Stable LAP detection algorithm

In order to detect a stable LAP $Q = (\phi, k, P_r)$ in the system, a set of process states satisfying Q needs to be found. To achieve this:

1. The set of states needs to form a consistent sub-cut. By running Algorithm 2 from Section 5, every process in the local BFST holds a local state which is part of a consistent sub-cut.
2. The predicate ϕ needs to be evaluated over the set of states. To achieve this, we execute Algorithm 3 which collects the set of states recorded in the consistent sub-cut, and then evaluates ϕ over this set.

Algorithm 3 is a convergecast within which the set of states recorded in the consistent sub-cut over the area of interest is collected at P_r using the tree edges in the local BFST. The convergecast uses the *State* message type. Each process P_i in the local BFST maintains the following variables:

- v_i: the variable(s) of the locally recorded snapshot state relevant to the evaluation of ϕ is/are stored;
- V_i: accumulates the snapshot states reported by processes in P_i's sub-tree within the local BFST;
- $\#children_i$: the number of children nodes in the local BFST; and
- $child_count_i$: the number of children from which a *State* message has been received.

Detection begins at leaf processes which have terminated Algorithm 2. These leaf processes in the local BFST initiates the convergecast by reporting the locally recorded state variable v_i to their parents in a *State* message. When an intermediate node P_i receives a *State* message, it accumulates the contained states from its sub-tree. When a *State* message has been received from all the children and Algorithm 2 has also terminated locally, P_i adds its own local snapshot state v_i to V_i. If P_i is not the initiator P_r, then P_i sends a *State*(V_i) message to its parent in the local BFST. However, if P_i was the initiator P_r, it evaluates the predicate ϕ over the set of states V. The algorithm is listed in Algorithm 3.

Adaptation to conjunctive LAP

The local variable v_i can be recorded as a Boolean for a conjunctive predicate. The local variable V_i that accumulates the states of processes within the sub-tree rooted at P_i can be represented as a Boolean to correspond to the (partial) evaluation of ϕ (over the sub-tree). This is because the evaluation of a conjunctive predicate is based on an *aggregation operation*, namely the AND operator.

- *Aggregation operations* are defined as those operations that are associative, thereby allowing the input to be processed in any order, and do not require all the input to be present before evaluation begins.

The size of a *State* message is thus $O(1)$.

Table 3
Complexity measures in degree-d bounded network.

Metric	Local BFST (Algorithm 1)	Consistent sub-cut (Algorithm 2)	Stable LAP aggregation based ϕ (Algorithm 3)	Stable LAP non-aggregation based ϕ (Algorithm 3)
Message count	$O(nkd)$	$O(n)$	$O(n)$	$O(n)$
Message size	$O(n^2)$	$O(n^2)$	$O(n)$	$O(nk)$
Storage cost	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Bandwidth cost	$O(n)$	$O(n)$	$O(1)$	$O(n)$

Algorithm 3 Stable LAP Detection Algorithm for $Q = (\phi, k, P_r)$ (code for P_i)

i. When P_i, which is a leaf node, terminates Algorithm 2:
1. send $State(\{v_i\})$ to parent; terminate;

ii. P_i receives $State(X)$ from child P_j:
2. $child_count_i = child_count_i + 1$;
3. $V_i = V_i \bigcup X$;
4. **if** ($child_count_i = \#children_i$) **then**
5. **await** (local termination of Algorithm 2);
6. $V_i = V_i \bigcup \{v_i\}$;
7. **if** $P_i \neq P_r$ **then** // non-root
8. send $State(V_i)$ to *parent* in local BFST; terminate;
9. **else**
10. evaluate $\phi(V_i)$; terminate.

Adaptation to relational LAP

The local variable V_i may be as large as the number of nodes in the sub-tree rooted at P_i, and hence varies from 1 to n. However, many relational predicates have their evaluations based on *aggregation operations*, e.g., addition, minimum, maximum, and average functions, and hence V_i is of size 1. Only for relational predicates whose evaluation is not based on *aggregation operations*, e.g., the median or the mode of a set of values, the size of V_i could be as large as n. The above observations on the size of V_i also hold for the size of the *State* message. The number of *State* messages is $n - 1$.

We propose a classification of predicates as follows:

- *Incremental predicate*: This is a predicate whose satisfaction can be determined for at least one input without evaluating the predicate fully over all variables.
  All conjunctive LAPs, such as Q_1, Q_2, Q_3 are incremental predicates. Some relational predicates that are non-conjunctive, such as Q_4, are also incremental predicates.
- *Non-incremental predicate*: This is a predicate whose satisfaction for every input can be determined only after evaluating it fully over all variables.
  Some relational predicates, such as Q_5, are non-incremental predicates.

Depending on the locally recorded snapshot values, the evaluation of relational predicates that are incremental may be determined without considering all the variables. If Algorithm 3 is modified to perform an iterative deepening breadth-first collection of the recorded v_i, i.e., a layer-by-layer reporting of the v_i's to P_r, the number of messages used is $\Omega(1)$; however, it will use $O(nk)$ number of messages.

7. Complexity analysis

We evaluate the complexities of our algorithms using four metrics:

- message count: count of the total number of messages generated by the algorithm.
- message size: the total size of all the control messages generated by the algorithm. It can be formalized as $\sum_i$ (*# type i messages $*$ size of type i messages*).
- storage cost: the space complexity at each process. Since P_r is responsible for storing the final results in Algorithm 3, we do not take P_r into consideration for this measure.
- bandwidth cost: bandwidth usage of a channel measures the total size of messages sent along that channel. The maximum bandwidth usage among all channels in the system is the bandwidth cost of the algorithm.

As mentioned before, all the algorithms share the scale-free feature. The complexities are evaluated within a degree-d bounded network. The results are shown in Table 3.

7.1. Algorithm 1 (local BFST)

Message count complexity: The local variable *dist* at each process in the area of interest can hold a value only within the range $[0, k]$. As per Observation 3, the local variable *dist* strictly decreases. Each time process P_i's *dist* decreases, it will send at most $d_i - 1$ *length* messages to its neighbors. Thus, each process can only send *length* messages to its neighbors up to k times. Each *length* message has one *ack* message. This gives a message count complexity of $O(k \sum_{i=1}^{n} d_i)$. In a degree-d bounded system, this is $O(nkd)$. Further, there are $n - 1$ *terminate* messages.

Message size complexity: The size of *length* control messages is $O(1)$. The sum of sizes of *ack* messages is bounded by nk^2d because the identifier of a node at distance j contributes to the T parameter j times. The size of a *terminate* is n. So the message size complexity is $O(nk^2d + n^2) = O(n^2)$.

Storage cost complexity: During the execution of Algorithm 1, each process has to maintain a *child_list* which is of $O(d)$ size. In addition, at the end of the algorithm, the root of the local BFST broadcasts the identifiers of all the processes in the local BFST using the *terminate* (T) message. This incurs an $O(n)$ storage cost at each process in the local BFST. So, the total storage cost is $O(n)$.

Bandwidth cost complexity: The value of each process' local variable *dist* can be only between 0 to k. Each time a process' local *dist* decreases, it will send a message to all its neighbors. Since *dist* strictly decreases, each process can only send up to k messages to each neighbor via the same channel. The size of *length* is $O(1)$. For the *ack* message, although its size is $O(n)$, the total size of *ack* messages sent by a single process is also $O(n)$ since no process identifier will be sent more than once. So the bandwidth cost is $O(n + k) = O(n)$.

7.2. Algorithm 2 (consistent sub-cut)

Message count complexity: The *INIT* message is broadcast within the local BFST only once, thus causing a $O(n)$ complexity. The modified *vector counter* technique uses another convergecast (*Cvg_Acc_White*) and broadcast (*Bcast_Acc_White*), thus causing another $O(n)$ complexity. So the total message count complexity is $O(n)$.

Message size complexity: The *INIT* control message has size $O(1)$. The two types of control messages – *Cvg_Acc_White* and *Bcast_Acc_White* – both have sizes of $O(n)$. So, the message size complexity is $O(n^2)$.

Table 4
Complexity comparison between LAP algorithms and some existing algorithms to detect stable predicates.

Metric	Message count	Message size	Storage	Bandwidth
Chandy–Lamport [4] (only snapshot recording considered)	$O(N^2)$	$O(N^2)$	$O(N)$	$O(1)$
Mattern's non-FIFO snapshot [28] (only snapshot recording considered)	$O(N)$	$O(N^2)$	$O(N)$	$O(N)$
LDP-Basic [11] (recording and predicate evaluation)	$O(d^{k-1})$	$O(kd^{k-1})$	$O(k)$	$O(d^{k-2})$
LDP-Snapshot [11] (recording and predicate evaluation)	$O(d^{2k-2})$	$O(kd^{2k-2})$	$O(k)$	$O(d^{k-1})$
Stable LAP initialization cost (Algorithm 1)	$O(nkd)$	$O(n^2)$	$O(n)$	$O(n)$
Stable LAP (aggregation based) (Algorithm 2 + 3)	$O(n)$	$O(n^2)$	$O(n)$	$O(n)$
Stable LAP (non-aggregation based) (Algorithm 2 + 3)	$O(n)$	$O(n^2)$	$O(n)$	$O(n)$

Storage cost complexity: Each process maintains a size-n vector $wmsg_sent_{l_i}$. So the storage cost is $O(n)$.

Bandwidth cost complexity: The *INIT* (of size $O(1)$), and the *Cvg_Acc_White* and *Bcast_Acc_White* messages (of size $O(n)$ each) traverse the edges of the BFST once. So the bandwidth cost is $O(n)$.

7.3. Algorithm 3 (stable LAP)

Recall that this algorithm works for aggregation based and non-aggregation based LAP. Aggregation based predicates include all conjunctive predicates and some non-conjunctive predicates. Non-aggregation based predicates include the remaining relational predicates.

Message count complexity: To detect the stable LAP, each process in the local BFST sends a *State* message to its parent in the BFST. So the message count complexity is $O(n)$. This result holds for aggregation based and non-aggregation based LAP.

Message size complexity: For aggregation based LAP, the *State* message, which is sent $n-1$ times, is of size $O(1)$. Thus, the message size complexity for aggregation based LAP is $O(n)$.

For non-aggregation based LAP, for $i \in [0, k-1]$, the d^{k-i} nodes in the local BFST that are $k-i$ hops away from P_r cumulatively send d^{k-i} *State* messages of size d^i to their parents. Thus, the cumulative size of all the *State* messages is $\sum_{i=0}^{k-1} d^{k-i}d^i = kd^k = kn$. Thus, the message size complexity for non-aggregation based LAP is $O(kn)$.

Storage cost complexity: For aggregation based LAP, the storage cost complexity is $O(1)$ because that suffices to store the local variables v_i, V_i, *num_children*, and *child_count*.

For non-aggregation based LAP, the space at P_i for V_i equals the number of nodes in the sub-tree rooted at P_i. At a distance i from P_r, this is d^{k-i}. In the worst case, this is $O(n)$.

Bandwidth cost complexity: One *State* message is sent on each local BFST channel for both aggregation based LAP and non-aggregation based LAP. The size of the *State* message is $O(1)$ for aggregation based LAP and $O(n)$ for aggregation based LAP. Hence, the bandwidth cost complexity for the two classes is $O(1)$ and $O(n)$, respectively.

7.4. Comparison with other algorithms

From the previous analysis, we can see that all the algorithms proposed in this paper are indeed scale-free. Taking the dominant complexity of those algorithms, we compare them with the complexities of some existing algorithms to detect stable predicates. Table 4 shows the result.

Notice that both Chandy–Lamport's and Mattern's snapshot algorithms have complexities that are affected by N. This is true even when using these algorithms to detect LAP because these algorithms do not have the features that a scale-free solution requires. Further, only the cost of snapshot recording is listed; Mattern's algorithm has an added cost of spanning tree or ring creation, and both these algorithms have another added cost of global state collection for predicate evaluation, that they do not address.

In a degree-d bounded network, where $n = d^{k-1}$, both *LDP-Basic* and *LDP-Snapshot* have a similar complexity compared to LAP algorithms. However, *LDP-Basic* and *LDP-Snapshot* cannot detect a predicate within a local region that is more complex than a *linear topology* and they cannot work in a system with non-FIFO channels or using multi-hop channels, as outlined earlier in Table 2.

The costs of our LAP algorithms can be split into two parts. (1) A one-time initialization cost (of Algorithm 1) for creating a BFST for predicate detection for $Q = (\phi, k, P_r)$. (2) The recurring cost incurred by Algorithms 2 and 3 each time the predicate Q needs to be evaluated.

8. Special cases

Repeated detection

The algorithms for detecting locality-aware predicates we introduced can also repeatedly detect LAP predicates within the same region. To achieve repeated detection within the same region, Algorithm 1 needs to run only once for constructing the local BFST. Each time a new detection begins, a new consistent sub-cut needs to be constructed. Hence Algorithm 2 will run repeatedly; we can alternate the roles of the two colors or cyclically use two from a set of three colors.

Weaker stable predicates

The locality-aware predicates we discussed so far target stable predicates. By detecting a weaker form of stable predicates, we can further cut down the algorithm complexity.

A weaker form of stable predicates, named as *strong stable* predicate, was defined in [31].

> A *strong stable predicate* is a stable predicate that, if true on some consistent cut, must remain true on all subsequent consistent cuts. Termination and deadlock are strong stable, though distributed garbage collection is not [31].

Detecting a strong stable predicate requires recording only consistent process states; observing the corresponding consistent channel states is not necessary. So, to detect a locality-aware strong stable predicate, we can simplify the second stage, namely that of constructing a consistent sub-cut. Applying only the *white/red coloring* technique is sufficient for detecting locality-aware strong stable predicates. This reduces the message size complexity of Algorithm 2 to $O(n)$, and its storage and bandwidth complexities to $O(1)$ (refer to Table 3). Consequently, the corresponding entries for strong stable LAP (aggregation-based) and strong stable LAP (non-aggregation based) in Table 4 will also reduce.

Modeling area of interest

If reduction of the initialization algorithm's complexity is needed, we can sacrifice the accuracy of modeling the area of interest. We can use a spanning tree instead of the local BFST, potentially leaving some processes within the area of interest unconsidered, to reduce the complexity of Algorithm 1. Comparing with the complexities of Algorithm 1 discussed in Table 3, we can achieve $O(nd)$ message count complexity and message size complexity, and $O(1)$ complexity in storage cost and bandwidth cost if we construct only a spanning tree. The changes to Algorithm 1 are as follows. In line (4), replace the test by "$dist = \infty$"; delete lines (7–8); in line (31), replace "BFST" by "ST".

9. Conclusions and future work

In this paper, we proposed the concept of a locality-aware predicate. This type of a predicate models a predicate in an area of interest. In a large-scale locality-driven network, such as modular robotics or WSNs, the following factors:

1. The interactions being local and driven by neighborhood proximity,
2. The high cost of doing a global predicate detection, and
3. The state of a local region better captures local interactions,

make locality-aware predicate detection a relevant and an interesting problem. We showed how to model the area of interest as a circular region (BFST rooted at P_r), and then proposed Algorithm 1 to efficiently construct the overlay network. This is the first distributed algorithm to construct a BFST within a local region in a graph. We also designed Algorithm 2 to efficiently take a snapshot within the area of interest in a non-FIFO network. This is the first algorithm to construct a consistent sub-cut defined by a region in a larger graph. We then defined two classes of locality-aware predicates: (1) conjunctive LAP, and (2) relational LAP. Finally, we gave an algorithm *Stable LAP Detection* for detecting both classes of LAPs, based on the recorded snapshot. The complexity analysis of all these algorithms showed their performance is *scale-free*. Hence, these algorithms have great potential for observing local regions within large-scale distributed systems such as modular robotics and WSNs.

There are several other potential extensions of locality-aware predicates, such as their adaptation to predicates of different detectability, e.g., unstable predicates [33]. We also plan to explore locality-aware predicate detection in a weighted graph, thus being able to specify more types of areas of interest. All these will broaden the application of locality-aware predicates.

References

[1] M. Ashley-Rollman, M. De-Rosa, S. Srinivasa, P. Pillai, S. Goldstein, Declarative programming for modular robots, in: IROS Workshop on Modular Robots, 2007, pp. 1–99.

[2] R. Atreya, N. Mittal, A. Kshemkalyani, V. Garg, M. Singhal, Efficient detection of a locally stable predicate in a distributed system, Journal of Parallel and Distributed Computing 67 (4) (2007) 369–385.

[3] Z. Butler, K. Kotay, D. Rus, K. Tomita, Generic decentralized control for a class of self-reconfigurable robots, in: IEEE International Conference on Robotics and Automation, 2002, pp. 809–816.

[4] K.M. Chandy, L. Lamport, Distributed snapshots: determining global states in distributed systems, ACM Transactions on Computer Systems 3 (1) (1985) 63–75.

[5] K.M. Chandy, J. Misra, Distributed computations on graphs: shortest path algorithms, Communications of the ACM (1982) 833–838.

[6] R. Cooper, K. Marzullo, Consistent detection of global predicates, in: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, 1991, pp. 163–173.

[7] C. Critchlow, K. Taylor, The inhibition spectrum and the achievement of causal consistency, in: Proc. the 9th ACM Symp. Principles of Distributed Computing, 1990, pp. 31–42.

[8] D. Darling, J. Mayo, X. Wang, Stable predicate detection in dynamic systems, in: Principles of Distributed Systems: 9th International Conference, 2006, pp. 161–175.

[9] J. del Prado Paven, S. Choi, Link adaptation strategy for IEEE 802.11 WLAN via recorded signal strength measurement, in: IEEE International Conference on Communications 2003 ICC'03, 2003, pp. 1108–1113.

[10] M. De-Rosa, S. Goldstein, P. Lee, J. Campbell, P. Pillai, Distributed watchpoints: debugging large modular robotic systems, International Journal of Robotics Research 27 (3) (2008) 315–329.

[11] M. De-Rosa, S. Goldstein, P. Lee, J. Campbell, P. Pillai, Detecting locally distributed predicates, ACM Transactions on Autonomous and Adaptive Systems 6 (2) (2011) 13:1–13:14.

[12] M. De-Rosa, S. Goldstein, P. Lee, P. Pillai, J. Campbell, Programming modular robots with locally distributed predicates, in: Proceedings of the IEEE ICRA, 2008, pp. 3156–3162.

[13] M. De-Rosa, S. Goldstein, P. Lee, P. Pillai, J. Campbell, A tale of two planners: modular robotic planning with LDP, in: 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2009, pp. 5267–5724.

[14] E. Dijkstra, C. Scholten, Termination detection for diffusing computations, Information Processing Letters (IPL) 11 (1980) 1–4.

[15] R. Garg, V. Garg, Y. Sabharwal, Efficient algorithms for global snapshots in large distributed systems, IEEE Transactions on Parallel and Distributed Systems 21 (5) (2010) 620–630.

[16] V.K. Garg, B. Waldecker, Detection of weak unstable predicates in distributed programs, IEEE Transactions on Parallel & Distributed Systems 5 (3) (1994) 299–307.

[17] V.K. Garg, B. Waldecker, Detection of strong unstable predicates in distributed programs, IEEE Transactions on Parallel & Distributed Systems 7 (12) (1996) 1323–1333.

[18] J. Helary, Observing global states of asynchronous distributed applications, in: Proc. the 3rd Int'l Workshop on Distributed Algorithms, 1989, pp. 45–56.

[19] G. Ho, C. Ramamoorthy, Protocols for deadlock detection in distributed database systems, IEEE Transactions on Software Engineering 8 (1982) 554–557.

[20] IEEE, IEEE 802.11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications, 2012.

[21] A. Kshemkalyani, Fast and message-efficient global snapshot algorithms for large-scale distributed systems, IEEE Transactions on Parallel and Distributed Systems 21 (9) (2010) 1281–1289.

[22] A. Kshemkalyani, M. Raynal, M. Singhal, An introduction to snapshot algorithms in distributed computing, Distributed Systems Engineering 2 (4) (1995) 224–233.

[23] A. Kshemkalyani, M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, Cambridge University Press, 2008.

[24] A. Kshemkalyani, B. Wu, Detecting arbitrary stable properties using efficient snapshots, IEEE Transactions on Software Engineering 33 (5) (2007) 330–346.

[25] T.-H. Lai, T. Yang, On distributed snapshots, Information Processing Letters (1987) 153–158.

[26] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.

[27] K. Marzullo, L.S. Sabel, Efficient detection of a class of stable properties, Distributed Computing 8 (2) (1994) 81–91.

[28] F. Mattern, Efficient algorithms for distributed snapshots and global virtual time approximation, Journal of Parallel and Distributed Computing 18 (4) (1993) 423–434.

[29] S. Peri, N. Mittal, Monitoring stable properties in dynamic peer-to-peer distributed systems, in: FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 2005, pp. 420–431.

[30] D. Rus, G. Chirikjian, Special issue on self-reconfiguring robots, Autonomous Robotics 10 (1) (2002) 1–99.

[31] A. Schiper, A. Sandoz, Strong stable properties in distributed systems, Distributed Computing 8 (2) (1994) 93–103.

[32] M. Shen, A. Kshemkalyani, A. Khokhar, Detecting tree distributed predicates, in: 2012 41st International Conference on Parallel Processing Workshops (ICPPW) Poster, 2012, pp. 598–599.

[33] M. Shen, A. Kshemkalyani, A. Khokhar, Detecting unstable conjunctive locality-aware predicates in large-scale systems, in: 2013 12th International Symposium on Parallel and Distributed Computing, ISPDC.

[34] K. Taylor, The role of inhibition in asynchronous consistent-cut protocols, in: Proc. the 3rd Int'l Workshop on Distributed Algorithms, 1989, pp. 280–291.

[35] J. Tsai, Flexible symmetrical global-snapshot algorithms for large-scale distributed systems, IEEE Transactions on Parallel and Distributed Systems 24 (3) (2013) 493–505.

Min Shen holds a B.S. in computer science from Nanjing University. He is currently a Ph.D. student in the Department of Computer Science at the University of Illinois at Chicago, advised by Prof. Ajay Kshemkalyani. His research interests include distributed algorithms, predicate detection and wireless sensor networks.

Ajay D. Kshemkalyani received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987, and the M.S. and Ph.D. degrees in computer and information science from Ohio State University in 1988 and 1991, respectively. He spent six years at IBM Research Triangle Park working on various aspects of computer networks, before joining academia. He is currently a professor in the Department of Computer Science at the University of Illinois at Chicago. His research interests are in distributed computing, distributed algorithms, computer networks, and concurrent systems. In 1999, he received the US National Science Foundation Career Award. He previously served on the editorial board of the Elsevier journal Computer Networks, and is currently an editor of the IEEE Transactions on Parallel and Distributed Systems. He has coauthored a book entitled Distributed Computing: Principles, Algorithms, and Systems (Cambridge University Press, 2008). He is a distinguished scientist of the ACM and a senior member of the IEEE.

Ashfaq Khokhar received his Ph.D. in computer engineering from the University of Southern California, in 1993. After his Ph.D., he spent two years as a Visiting Assistant Professor in the Department of Computer Sciences and School of Electrical and Computer Engineering at Purdue University. In 1995, he joined the Department of Electrical and Computer Engineering at the University of Delaware, where he first served as Assistant Professor and then as Associate Professor. In Fall 2000, Dr. Khokhar joined UIC in the Department of Computer Science and Department of Electrical and Computer Engineering, where he currently serves with the rank of Professor. Dr. Khokhar has published over 200 technical papers and book chapters in refereed conferences and journals in the areas of wireless networks, multimedia systems, data mining, and high performance computing. He is a recipient of the NSF CAREER award in 1998. His paper entitled "S-to-P Broadcasting in Message Passing MPPs" has won the Outstanding Paper award in the International Conference on Parallel Processing in 1996. He has served as the Program Chair of the 17th Parallel and Distributed Computing Conference (PDCS), 2004, Vice Program Chair for the 33rd International Conference on Parallel Processing (ICPP), 2004, and Program Chair of the IEEE Globecom Symposium on Ad hoc and Sensor Networks, 2009. He is a Fellow of the IEEE for his contributions to multimedia computing and databases. His research interests include: wireless and sensor networks, multimedia systems, data mining, and high performance computing.