Detecting Locally Distributed Predicates

MICHAEL DE ROSA, SETH COPEN GOLDSTEIN and PETER LEE
Carnegie Mellon University
JASON CAMPBELL and PADMANABHAN S. PILLAI
Intel Labs, Pittsburgh

In this paper, we formalize *locally distributed predicates*, a concept previously introduced to address specific challenges associated with modular robotics and distributed debugging. A locally distributed predicate (LDP) is a novel construction for representing and detecting distributed properties in sparse-topology systems. Previous work on LDPs presented empirical validation; here we show a formal model for two variants of the LDP algorithm, LDP-Basic and LDP-Snapshot, and establish performance bounds for these variants. We prove that LDP-Basic can detect strong stable predicates, that LDP-Snapshot can detect all stable predicates, and discuss their applicability to various distributed programming domains and to spatial computing in general. LDP detection in bounded-degree networks is shown to be scale-free, making the approach particularly attractive for specific topologies even though LDPs are less efficient than snapshot algorithms in general distributed systems.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*

General Terms: Languages, Theory

Additional Key Words and Phrases: distributed predicates, distributed computing, snapshots, consistency

1. INTRODUCTION

Distributed systems have had a long and active history in the research and scientific computing communities. Recently, they have found widespread acceptance in such applications as modular robots [Rus and Chirikjian 2001; Butler et al. 2002], pervasive environmental sensing [Levis et al. 2004], and large-scale wireless networks [Karp and Kung 2000]. These applications are distinguished by their highly dynamic topologies, and their large size, numbering in the thousands of processes. The emergence of such massive and dynamic systems has led to a need for robust, scalable algorithms to program, manage, and reason about distributed processes.

The same qualities of scale and dynamism that make distributed systems exciting also pose certain challenges. Fundamentally, the difficulties of programming

Author's address: M. De Rosa, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2009 ACM 0164-0925/2009/0500-0001 $5.00

distributed systems stem from *distributed state*. For a system to be distributed, it must in some way make use of the state of multiple processes, or else it is just a number of disconnected computations. Making use of distributed state requires transferring state information over communications links between processes, which introduces delay and potential message loss. If the processes in a distributed system are not centrally coordinated, then there may be no common notion of time shared by all processes. If there is no common time-base, then what does it mean to say that processes P_i and P_j are in the same state? The issue of *asynchronicity* makes reasoning about distributed systems difficult.

Without a global clock, the notion of the state of the entire distributed system becomes ill-defined, as there is no shared moment in time that all processes can be examined. In his seminal paper, Lamport [1978] described the *happens-before* relationship imposed by causality. Briefly, the happens-before relationship states that, for a system to be consistent with causality, the sending of a message must precede its reception (at all observers). This relationship can be used to create a partial-ordering on a set of distributed events, and thus show whether or not a particular view of a distributed system is *consistent* with causality. This observation by Lamport led to a large number of algorithms for dealing with distributed state, relying on the use of the happens-before relation to ensure consistency.

Interest in proving properties of distributed systems led to the creation of many *distributed predicate detection* algorithms. Properties such as deadlock [Obermarck 1982] and termination [Tel and Mattern 1993] can be expressed as predicates over the state of distributed processes. The complete state of the distributed system can then be gathered and evaluated, to determine if the predicate is true. However, there is a problem with such a naive approach. The gathered state creates an implicit serialization of the parallel event streams in the system. There may be multiple consistent serializations, and the predicate may be true in some of them, but not others. This leads to a situation where predicates can be *possibly-true* (true in at least one consistent serialization) or *definitely-true* (true in all consistent serializations) [Cooper and Marzullo 1991].

To eliminate the need to explore an exponentially-bounded number of serializations, much of the research in distributed predicate detection has been targeted at the class of *stable predicates* [Babao et al. 1993]. Stable predicates are distributed predicates that, once evaluated as true under any consistent serialization, will remain true for all subsequent serializations. The classical algorithm for detecting such predicates is the Chandy-Lamport snapshot algorithm [Chandy and Lamport 1985]. There have been many subsequent algorithms [Kshemkalyani et al. 1995] focused either on improving the performance of distributed snapshots, or on delineating subclasses of stable predicates with more tractable properties.

LDPs are differentiated from more traditional classes of distributed predicates in that they are concerned with topology, rather than state. They form an orthogonal categorization schema to stable properties and their subclasses. Specifically, in systems where there is a sparse communications topology, and topology is important, it is our assertion that locally distributed predicates are a more natural and efficient way to model and detect distributed properties.

This paper begins with a brief review of distributed systems and common dis-

tributed predicate classes, before describing the characteristics of LDPs. We then present two algorithms for detecting important subclasses of locally distributed predicates, including proofs of correctness and complexity measures. We close with a discussion of the utility of LDPs, as demonstrated in previous work on distributed debugging [De Rosa et al. 2008] and programming modular robots [De Rosa et al. 2009].

2. PRELIMINARIES

A *distributed system* is a graph (P, L), where P is the set of processes (nodes) and L is the set of communication links (edges) connecting them. Let $n =\mid P \mid$ and $l =\mid L \mid$.

The execution history at process $P_i \in P$ is the alternating sequence of states and events $\langle s_i^0, e_i^1, s_i^1, e_i^2, s_i^2, e_i^3, \cdots \rangle$, where s_i^0 is the initial state of the process and e_i^k is the k^{th} event at that process. An *event* at a process can be message reception, message transmission, or an internal event. We denote an arbitrary state from P_i as s_i^*, and the current state (observed locally by P_i) as s_i.

The state of a process is composed of an application-dependent number of named state variables, with numeric or boolean values. The value of state variable *var* at the k^{th} state of process P_i is denoted by var_i^k, and the current value of a particular state variable is likewise denoted var_i.

A *channel* C_{ij} is a reliable, unidirectional FIFO link between processes P_i and P_j, with a finite (but not bounded) delay. For two processes P_i and P_j, the channel C_{ij} exists if $L_{ij} \in L$. The graph is undirected, thus $L_{ij} \in L \Rightarrow L_{ji} \in L$. We say that two processes P_i, P_j are *neighbors* if $L_{ij} \in L$. The state of a channel $SC_{ij} = transit(e_i^{x_i}, e_j^{x_j})$ is the set of messages sent by P_i up to event $e_i^{x_i}$ and not received by P_j up to event $e_j^{x_j}$. We denote the k^{th} message sent along C_{ij} from P_i to P_j as m_{ij}^k.

An *execution* of a distributed system with n processes is $(E, \prec)$, where $\prec$ is the Lamport happens-before relation on E, the set of events $E = \bigcup_{i \in P} e \in E_i$. E_i is the totally ordered event sequence at process P_i.

A *consistent cut* K is a subset of E such that if $e \in K$ then $(\forall e')\ e' \prec e \Rightarrow e' \in K$. We can define a *consistent sub-cut* of a distributed system as a consistent cut over some subset of P. To show that a sub-cut K is consistent, it is sufficient to show:

(1) there is no *orphan* message m_{ij} such that $receive(m_{ij}) \in K \wedge send(m_{ij}) \notin K$

(2) if $e_i^k \in K$, then $(\forall j < k) e_i^j \in K$

Less formally, a sub-cut K is consistent if all message reception events in K have matching message transmissions, and the events of P_i included in K form a prefix of the execution history at P_i [Raynal 1999]. K_2 is subsequent to K_1 iff all $e \in K_1$ are also in K_2, and for some $e_2 \in K_2$, $\exists e_1 s.t. e_1 \prec e_2$.

3. CLASSICAL DISTRIBUTED PROPERTIES

The field of distributed predicate detection is a well-established and active one. There are a large number of algorithms and predicate types described in the literature. We briefly summarize several of the more important classes of predicates,

deferring a more thorough categorization to several excellent survey papers on the topic [Kshemkalyani et al. 1995; Baldoni and Raynal 2002].

stable: A predicate that, once found to be true by a global snapshot, will remain true [Chandy and Lamport 1985]. This class of predicate explicitly includes channel state, and can be used to detect deadlock, termination, and the number of tokens in a token-passing system.

strongly stable: A stable predicate that, if true on some cut (consistent or not), must remain true on all subsequent cuts [Lai and Yang 1987]. Deadlock is not strongly stable.

strong stable: A stable predicate that, if true on some consistent cut, must remain true on all subsequent consistent cuts [Schiper and Sandoz 1994]. Termination and deadlock are strong stable, though distributed garbage collection is not.

locally stable: A stable predicate such that, once it becomes true, none of the variables used by the predicate change in value [Marzullo and Sabel 1994]. Termination, deadlock, and global virtual time are locally stable, determining the number of tokens in a token-passing system is not.

conjunctive stable: A predicate formed from the conjunction of local predicates, each of which is individually stable [Fidge 1991; Raynal 1999].

Strongly stable, strong stable, locally stable, and conjunctive stable are all subclasses of stable predicates. Conjunctive stable predicates are a subclass of strong stable predicates. There is no proven relationship between the classes of strong stable and locally stable, though Schiper and Sandoz speculated that they were similar.

Each of these classes of predicates is differentiated by the requirements they place on detection algorithms: the presence/absence of channel state, the need for consistent cuts, and the ways in which process state may change once a predicate becomes true. Algorithms for detecting various classes of stable predicates are distinguished by the particular class of predicates they detect, the type of communication channel they assume (FIFO, non-FIFO, or causal delivery), and the degree to which they interfere with existing message traffic.

4. LOCALLY DISTRIBUTED PREDICATES

Locally distributed predicates are distinguished from most classical categories of properties in that they are defined by their topology, rather than their detectability. An LDP is a distributed predicate over a *fixed-size, linearly-connected* group of of processes. A set of processes that satisfies the LDP must form a connected chain, with communication channels linking each of them. This formulation allows LDPs to easily express communication distance (number of hops) and other simple topological constraints, such as adjacency (whether of not two processes share a channel), shared adjacency (whether or not two processes share a neighbor), etc. Note that an LDP is limited to describing predicates that can be expressed in terms of a fixed number of linearly-connected processes. This includes chains of processes and rings, but excludes trees and other arbitrary DAGs.

4.1 Formal Properties

A *locally distributed predicate* (LDP) is $Q(a_1, a_2, \cdots, a_k)$, a predicate over the state of k abstract processes. Let $\mid Q \mid = k$ be the size of the predicate. We say that Q is *satisfied* by V if, for some vector of states $V = \langle s^*_{i_1}, s^*_{i_2}, \cdots \rangle$from distinct processes, where $\mid V \mid \leq k$:

(1) $L_{i_1 i_2} \in L, L_{i_2 i_3} \in L, \cdots$ for all pairs of P which are adjacent in V
(2) $Q \leftarrow V =$ `true`, where $\leftarrow$ is element-wise state substitution of V into Q

We will refer to these two properties as *adjacency* and *substitution*, respectively. Adjacency requires that, if $s^*_{i_1}$ and $s^*_{i_2}$ are adjacent in the state vector V, then their respective processes P_{i_1} and P_{i_2} must be neighbors. Taken as a whole, this requires that the processes represented in V form a linear, connected chain via their communications channels.

4.2 State Substitution Operator

Substitution is the second requirement for LDP satisfaction. An LDP is expressed in terms of abstract processes, creating a predicate from their state and imposing a restriction on their valid topology. Take the simple LDP:

$$Q(a_1, a_2) = flag_{a_1} \wedge flag_{a_2}$$

This is a predicate over two abstract processes, a_i and a_2, which must be neighbors (due to adjacency). Both a_1 and a_2 have a boolean state variable $flag$, whose value must be true. Note that this formulation explicitly omits any reference to consistency or detectability, it is concerned only with topology.

To instantiate this LDP over the state of two processes P_i and P_j, we first construct a vector with their state:

$$V = \langle s^*_i, s^*_j \rangle$$

Element-wise state substitution yields:

$$(flag_{a_1} \wedge flag_{a_2}) \leftarrow \langle s^*_i, s^*_j \rangle = (flag^*_i \wedge flag^*_j)$$

Where a_1 has been instantiated as P_i and a_2 has been instantiated as P_j. If this predicate is true, and P_i and P_j are neighbors, then Q is satisfied by P_i and P_j.

Once state substitution has been performed, the predicate can be evaluated to produce one of three output values: `true`, `false`, and `undefined`. As $\mid V \mid \leq \mid Q \mid$ in $Q \leftarrow V$, there may be instances where the predicate can not be definitively evaluated as either `true` or `false`. In this case the predicate's value is `undefined`

4.3 Example LDPs

Using LDPs, it is easy to formulate topology-limited versions of many classical distributed properties. One need merely decide on the number of abstract processes, and state the property in terms of those processes. For example, the termination of two adjacent processes is represented as:

$$Q(a_1, a_2) = terminated_{a_1} \wedge terminated_{a_2}$$

While the presence of more than 1 token in a 3-process ring is:

$$Q(a_1, a_2, a_3) = token_{a_1} + token_{a_2} + token_{a_3} > 1$$

Process P_i begins detection of Q:
if $Q \leftarrow \langle s_i \rangle = true$ **then**
  return $\langle P_i \rangle$ as matching Q
else
  $m = \langle s_i \rangle$;
  send m to all neighbors of P_i;
end

Process P_j receives message $m = \langle s^*_{i_1}, s^*_{i_2}, s^*_{i_3}, \cdots \rangle$:
$V = \langle s^*_{i_1}, s^*_{i_2}, s^*_{i_3}, \cdots s_j \rangle$;
if $Q \leftarrow V = true$ **then**
  return $\langle P_{i_1}, P_{i_2}, P_{i_3}, \cdots, P_j \rangle$ as matching Q
else
  if $\mid V \mid < \mid Q \mid$ **then**
    $m' = V$;
    send m' to all neighbors of P_j not already represented in V;
  else
    failed to match;
  end
end

Algorithm 1: LDP Detection (Non-Inhibitory)

Finally, mutual deadlock of two neighboring processes is:

$$Q(a_1, a_2) = (waitingOn_{a_1} == id_{a_2}) \wedge (waitingOn_{a_2} == id_{a_1})$$

It is obvious that the detectability of the above LDPs is consistent with their classical categorization. In other words, though all of the examples are locally distributed predicates, they also fall into categories such as stable, strong stable, etc. We can therefore define a strong stable LDP, for example, as a locally distributed predicate that, once it becomes satisfied for a particular subset of processes, remains satisfied for that subset.

5. THE LDP-BASIC ALGORITHM

To detect subgroups that match a given LDP, we have developed a simple distributed search algorithm. This algorithm, which we call LDP-Basic, gathers a state vector to be substituted into a given LDP. We show that the state gathered by LDP-Basic is a consistent sub-cut of the system. From Schiper and Sandoz [1994], we know that this property allows LDP-Basic to detect properties that are strong stable, an important subclass of stable properties.

5.1 Algorithm

Detection begins at the initiator process P_i, where the current local state is substituted into Q. If this substitution evaluates to true, then P_i alone satisfies the LDP. If not, the 1-element vector $\langle s_i \rangle$ is sent to all neighbors of P_i.

When a message containing a state vector is received at process P_j, the process appends its current state s_j to the vector, and performs substitution into Q. If this substitution evaluates to true, then the processes represented by the state vector satisfy Q. If not, the size of the vector is compared to the number of processes specified by Q. If $\mid V \mid < \mid Q \mid$, then the state vector V is sent to all neighbors of P_j

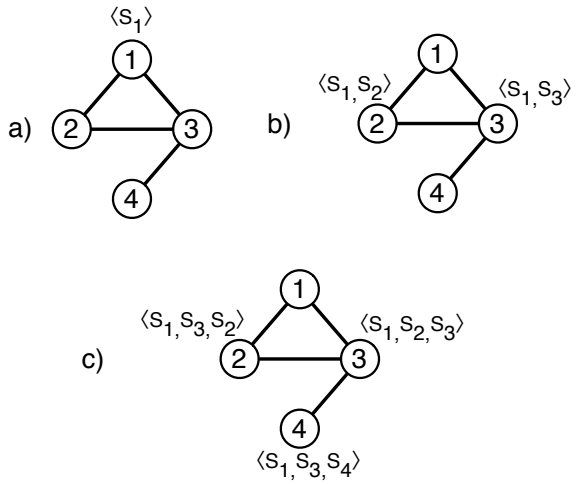

Fig. 1. Progression of the LDP-Basic algorithm from initiation at P_1 (a), through two phases of spreading (b,c). Processes are labeled circles, with messages at those processes located adjacent to them.

not already represented in V. In this way, we sequentially gather the current state from P_{i_1}, then P_{i_2}, etc.

Figure 1 illustrates the operation of LDP-Basic over a predicate Q of size 3. In Figure 1a, process 1 has created a vector containing its current state. In Figure 1b, that state has been transmitted to processes 2 and 3, which have appended their current state to the vector, and so on in Figure 1c. If any other these vectors satisfied Q, that condition would have been detected at the last process necessary to fulfill the predicate. Note that the algorithm supports multiple overlapping initiators, and will correctly identify multiple overlapping subgroups that satisfy Q, identifying them by their distinct vector of processes.

5.2 Consistency Proof

To show that the state captured by this algorithm is a consistent sub-cut, we must show that all states that are captured obey the Lamport happens-before relation. There are two components to this proof. The first is to show that if $e_i^k \in C$, then $(\forall j < k) e_i^j \in C$. This is trivially true, as s_i^k encompasses the effect of $(\forall j < k) e_i^j$ by definition. Next, we must show that there are no orphan messages within the sub-cut. This requires the use of the following lemma:

LEMMA 1. *Under LDP-Basic, if state vector V contains $s_i^{x_i}$ and $s_j^{x_j}$, where $s_i^{x_i}$ appears in V before $s_j^{x_j}$, then $s_i^{x_i} \prec s_j^{x_j}$.*

PROOF. Process state is only appended to V. Each process appends only one element, its current state. If process P_j appends $s_j^{x_j}$ to V while $s_i^{x_i}$ is already a member of V, then P_j must have received V (and thus $s_i^{x_i}$) from another process. Thus, by the causality of message delivery, $s_i^{x_i} \prec s_j^{x_j}$. □

THEOREM 5.2.1. *The state vector gathered by LDP-Basic is a consistent sub-cut.*

Process P_i begins detection of Q:
Obtain an exclusive lock on spanning subtree T_i, of depth $\mid Q \mid$;
Send $\langle Suppress, T_i \rangle$ to $c(T_i, P_i)$;

Process $P_j \in T_i$ receives $\langle Suppress, T_i \rangle$:
Suppress transmission of any message not related to P_i's detection attempt ;
Send $\langle Suppress, T_i \rangle$ to all neighboring $P_k \in T_i$;
Once P_j has received $\langle Suppress, T_i \rangle$ from all neighboring $P_k \in T_i$, send $\langle Ready, P_j \rangle$to P_i;

Process P_i receives $\langle Ready, T_i \rangle$ from all $P_j \in T_i$:
P_i begins Algorithm 1 ;
P_i releases lock on subtree T_i ;

Process $P_j \in T_i$ is unlocked from T_i:
Resume transmission of any message not related to P_i's detection attempt ;

Algorithm 2: LDP Detection (Inhibitory)

PROOF. By contradiction: Take any two states from V, $(s_i^{x_i}, s_j^{x_j})$ where $s_i^{x_i}$ appears before $s_j^{x_j}$. Suppose that there is an orphan message m_{ji}, such that $receive(m_{ji})$ is known at $s_i^{x_i}$, but $send(m_{ji})$ is not known at $s_j^{x_j}$. From Lemma 1, $s_i^{x_i} \prec s_j^{x_j}$. From the causality of message delivery, $send(m_{ji}) \prec receive(m_{ji})$. From the assumption that m_{ji} is an orphan, $receive(m_{ji}) \prec s_i^{x_i}$ and $s_j^{x_j} \prec send(m_{ji})$. As $\prec$ is commutative, $s_j^{x_j} \prec receive(m_{ji})$, and thus $s_j^{x_j} \prec s_i^{x_i}$. This contradicts $s_i^{x_i} \prec s_j^{x_j}$, and thus there can be no such message m_{ji}. □

This proof shows that any sub-cut produced by LDP-Basic is a consistent one; from Schiper and Sandoz, we know that this means that LDP-Basic can detect strong properties, including the sub-categories of conjunctive stable and (potentially) locally stable properties. This includes such properties as termination, deadlock, and virtual time, when the properties are restricted to finite chains of processes.

6. DETECTING STABLE LOCALLY DISTRIBUTED PREDICATES

The LDP-Basic algorithm is sufficient to detect LDPs in the class of strong stable predicates, but it will not detect the more general stable LDPs. In particular, LDP-Basic cannot detect predicates whose satisfiability may depend on channel state. We therefore introduce a second algorithm, which can detect a broader range of predicates.

6.1 Algorithm

Our second algorithm, LDP-Snapshot, is an extension of the basic detection method, and allows for the capture of all stable LDPs. It does so by coercing sub-graphs of the system into configurations where there is no channel state, and then performing the basic detection operation. By ensuring that all channels are empty, any stable LDP must reside purely in process state, and thus be observable via a consistent sub-cut. We model our work on on the inhibitory algorithms of Critchlow and Taylor [1990], which used a similar technique to create consistent cuts in non-FIFO systems.

LDP-Snapshot operates as follows: Let T_i be the spanning subtree of depth $\mid Q \mid$ rooted at P_i. It is obvious that all vectors of processes that match Q and begin

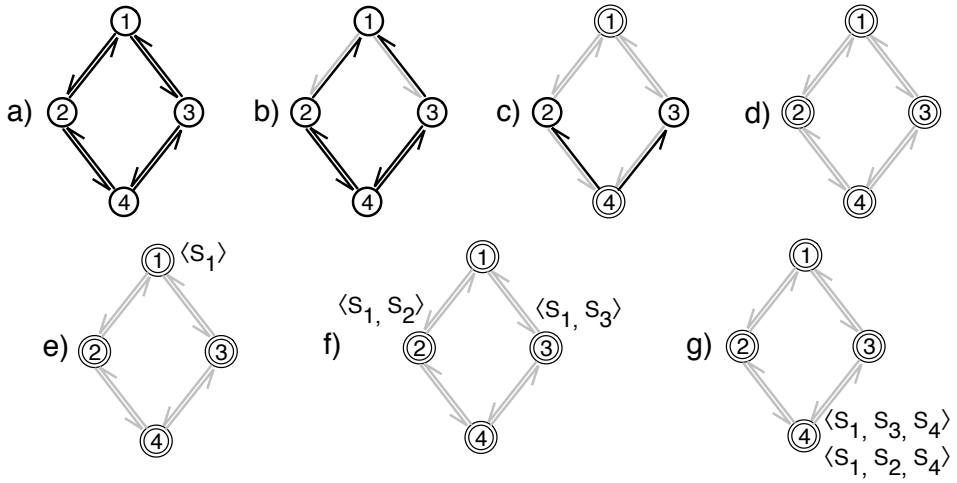

Fig. 2. Progression of the LDP-Snapshot algorithm from initiation at P_1 (a) Initial state of T_1, with depth 3. (b) P_1 sends ⟨Suppress⟩ to its neighbors, and suppresses its outbound channels (represented in grey). (c) P_1 has received a suppression message from all neighbors in T_1, and is now ⟨Ready⟩, represented by a double outline. (d) All processes have sent ⟨Ready⟩ messages. (e) Process P_1 initiating LDP-Basic, showing local state vector. (f,g) Continuation of LDP-Basic, showing state vectors growing.

in P_i must be contained within T_i. To obtain a snapshot, P_i begins by obtaining an exclusive lock on all processes in T_i, to prevent overlapping snapshots. Once the lock has been obtained, P_i distributes the message $\langle Suppress, T_i\rangle$ downwards through the tree, as seen in Figure 2. Once Suppress has been sent via a channel, transmission via the channel is delayed until after the snapshot is taken. Once a process has received Suppress messages from all neighbors that belong to T_i, it sends $\langle Ready, T_i\rangle$to P_i via the spanning tree. Once P_i has received Ready messages from all members of T_i, it proceeds with LDP-Basic, and then unlocks the subtree.

6.2 Snapshot Consistency Proof

From Chandy and Lamport [1985], we know that distributed snapshots require a consistent set of process and channel state. In LDP-Snapshot, we first empty and suppress as communications channels within a subtree, and then take a consistent cub-cut of the process state in that subtree. Consistency of the sub-cut is obtained via the use of LDP-Basic, whose properties are described in Section 5. Proving that all channels are empty is as follows:

THEOREM 6.2.1. *When LDP-Basic is executed by P_i, $SC_{jk} = \emptyset, \forall (P_j, P_k) \in T_i$.*

PROOF. Once a process P_j receives $\langle Suppress, T_i\rangle$ from P_k, it is guaranteed that $SC_{kj} = \emptyset$, and that all further messages along C_{kj} will be inhibited. A process P_j will not send $\langle Ready, P_j\rangle$ until it has received $\langle Suppress, T_i\rangle$ from all neighbors in T_i. Therefore, sending $\langle Ready, P_j\rangle$ guarantees that $\forall P_k \in T_i, SC_{kj} = \emptyset$. Once the initiator P_i has received $\langle Ready, P_j\rangle$ from all $P_j \in T_i$, then $\forall (P_j, P_k) \in T_i, SC_{jk} = \emptyset$. Therefore, when the basic LDP detection algorithm runs after P_i has received $\langle Ready, P_j\rangle$ from all $P_j \in T_i$, it will be guaranteed that $SC_{jk} = \emptyset, \forall (P_j, P_k) \in$

Table I. Complexity Measures of Detection Algorithms (Degree-d Topology)

Metric	C-L	LDP-Basic	LDP-Snapshot
Messages (Single Initiator)	$O(n^2)$	$\boldsymbol{O(d^{k-1})}$	$\boldsymbol{O(d^{2k-2})}$
Messages (n Initiators)	n/a	$O(nd^{k-1})$	$O(nd^{2k-2})$
Bandwidth	$O(n)$	$\boldsymbol{O(d^{k-2})}$	$\boldsymbol{O(d^{k-1})}$
Storage	$O(n)$	$\boldsymbol{O(k)}$	$\boldsymbol{O(k)}$

Table II. Complexity Measures of Detection Algorithms (Arbitrary Topology)

Metric	C-L	LDP-Basic	LDP-Snapshot
Messages (Single Initiator)	$O(n^2)$	$O(n^{k-1})$	$O(n^{k-1})$
Messages (n Initiators)	n/a	$O(n^k)$	$O(n^k)$
Bandwidth	$O(n)$	$O(n^{k-2})$	$O(n^{k-2})$
Storage	$O(n)$	$\boldsymbol{O(k)}$	$\boldsymbol{O(k)}$

T_i. □

As channel state is empty, the state captured by LDP-Basic is a consistent capture of both channel and process state, and thus a distributed snapshot. From Chandy and Lamport, LDP-Snapshot is thus able to detect all properties in the class of stable LDPs.

7. PERFORMANCE AND COMPLEXITY

In evaluating our algorithms for detecting LDPs, we will show complexity results for three important metrics: overall message traffic, maximum channel bandwidth, and required state storage. We compare our algorithms to the classic Chandy-Lamport distributed snapshot algorithm. While there are more efficient snapshot algorithms [Kshemkalyani and Wu 2007; Venkatesan 1989], the Chandy-Lamport algorithm is both widely known and very succinct.

We present upper bounds for systems of bounded degree (Table I), which are reflective of the sparse-topology systems that LDPs were designed for. Additionally, we present upper bounds for general graphs (Table II). Scale-free properties are indicated by bold text in both tables. In all of our analyses, the number of processes is n, the maximum degree of the system is d, and the size of the relevant LDP is k.

7.1 Messaging Cost

Message complexity is measured by assigning a cost of 1 to each message transmission. We assume that messages are not merged or batched in any way, though they would be in most concrete implementations. In a similar vein, the cost of a message carrying a single boolean value, and one containing the entire state of a process are both 1.

The base-line Chandy-Lamport snapshot algorithm has a message complexity of $O(n^2)$, dominated by two factors. The first is the need to to send a marker message along each channel (of which there are potentially $O(n^2)$). The second is the need to aggregate the state of each process at some distinguished initiator, which involves n processes sending their recorded state to the initiator, along paths that can be $O(n)$ hops long.

LDP-Basic's messaging complexity is determined by the spread of the state vector from the initiator, to each of its neighbors, to each of their neighbors, etc. for up to k processes. In a fully connected system, each process has $O(n)$ neighbors, giving a cost of $O(n^{k-1})$ for a detection of a size-k LDP starting at some process P_i. Starting a detection attempt at every process raises the total cost to $O(n^k)$ messages.

The cost of LDP-Snapshot, exclusive of the application-dependent cost of acquiring a lock on T_i, is determined by three factors: the cost to send suppression messages along each channel in T_i, the cost for all members of T_i to send Ready messages back to P_i, and the cost to run LDP-Basic once. These costs are $O(\mid T_i \mid^2)$ (for any spanning tree), $O(\mid T_i \mid^2)$ (for the worst -case spanning tree), and $O(n^{k-1})$. As the maximum size of a depth-k spanning tree is n, these costs are $O(n^2)$, $O(n^2)$, and $O(n^{k-1})$ respectively, for a total complexity of $O(n^{k-1})$ for an attempt rooted at a given P_i. The complexity for each process to initiate the algorithm is $O(n^k)$ messages.

In a system with a maximum degree of d, these metrics change in important ways. The complexity of Chandy-Lamport is still $O(n^2)$, as aggregating state at a central location is an $O(n)$ operation for each process. The cost of LDP-Basic becomes dependent on the maximum degree d, leading to a cost of $O(d^{k-1})$ for a single initiator, or $O(nd^{k-1})$ for all processes to be initiators. It is important to note that $O(d^{k-1})$ is a constant for a given predicate and maximum degree. Similarly, as the maximum size of a depth-k subtree becomes d^{k-1} in a system with maximum degree d, the cost of LDP-Snapshot becomes $O(d^{2k-2})$ for a single initiator, or $O(nd^{2k-2})$ for n initiators.

7.2 Bandwidth

We measure bandwidth complexity by observing the number of messages sent along each channel, and taking the maximum over all channels as the complexity. Chandy-Lamport has a bandwidth complexity of $O(n)$, in the case where the initiator process has only a single channel connecting it to the remainder of the system. In that case, all $n-1$ processes must route their state along this edge.

The bandwidth complexity of LDP-Basic can be determined as follows: assume that the channel C_{ab} is the channel with the highest bandwidth. Furthermore, assume that P_b is located at distance $k-1$ from initiator P_i. The bandwidth through P_{ab} is then the number of distinct paths of length $k-2$ from P_i to P_a, as each of these paths will produce a distinct state vector, which must be forwarded to P_b. The maximum bandwidth is thus $O(n^{k-2})$ for arbitrary systems, or $O(d^{k-2})$ for systems with a maximum degree of d.

The maximum bandwidth needed for LDP-Snapshot is derived from two terms: the cost of every member of T_i sending a Ready message to P_i, and the cost of executing LDP-Basic. Using similar reasoning to Chandy-Lamport, the maximum bandwidth for each member of T_i to send a Ready message to P_i is $O(n)$ in an

arbitrary system, or $O(d^{k-1})$ in a degree-d system. The maximum bandwidth for LDP-Snapshot is therefore $O(n^{k-2})$ for arbitrary systems, or $O(d^{k-1})$ for degree-d systems.

7.3 State Storage

Once a snapshot or sub-cut has been initiated, the state of the involved processes must be gathered and processed in some manner, so that the satisfiability of the distributed predicate can be evaluated. In Chandy-Lamport, the distinguished initiator must gather state from every process, requiring $O(n)$ space to store it before the global predicate can be evaluated. In the case of both LDP-Basic and LDP-Snapshot, state is carried in a vector message, and each vector is processed individually, requiring a maximum of $O(k)$ storage at any given process.

7.4 Optimization

The presented algorithms for LDP-Basic and LDP-Snapshot are deliberately simple, and there is much room for improvement. As a simple optimization, replace the condition

$$\text{if } \mid V \mid < \mid Q \mid$$

on line 13 of LDP-Basic (Alg.1) with

$$\text{if } (\mid V \mid < \mid Q \mid) \vee \ (Q \leftarrow V = \text{false})$$

This optimization enables aggressive Boolean short-circuiting in LDPs. That is to say that, if a state vector fails state substitution, then that vector, and all subsequent vectors that it would generate, are removed from consideration early. In previous work on distributed debugging, we showed that this optimization provided significant benefit for predicates that failed early, or matched infrequently.

7.5 Mitigation of Cost Growth

An important shortcoming of the Chandy-Lamport approach is that as the system is scaled up, the costs of running the algorithm grow. As these costs depend only on n, there is nothing a system designer can do to control them. With LDPs, these costs depend instead on the degree of the connectivity graph and the complexity of the predicate and not, in general, on n. Hence, there are parameters available to the system designer, which allow some control of the costs of running distributed predicate detection, even as the system size grows. For example, limiting the degree of connectivity between nodes or the length of predicates allowed can keep algorithmic complexity low despite a large value of n.

8. LDP APPLICATIONS

Because messaging costs for LDP-Basic and LDP-Snapshot grow exponentially with the degree of a system's connectivity graph, LDPs are best suited to large-scale distributed systems with sparse connectivity, and especially when computation and communication patterns are driven by network locality. Systems of this type include wireless sensor networks, modular robots, and large-scale WLANs.

An obvious use for LDPs is in distributed monitoring and debugging. By specifying predicates corresponding to incorrect or important state, predicates can flag

the presence of the errors and other conditions. This is especially important in the case of distributed errors, which manifest across the state of multiple processes, and are thus not amenable to traditional debugging/logging techniques. As an example, take the problem of distributed leader election. Determining the presence of multiple leaders is, in many cases, impossible given the state of just one process. There are many similar properties, e.g. the problem of disconnection detection in modular robotics. For a more comprehensive treatment of distributed debugging, utilizing a precursor to locally distributed predicates, please see our previous work on the subject [De Rosa et al. 2008].

LDPs can also form the basis for a distributed, declarative programming system. By binding an action to the detection of a given predicate, we can create a simple declarative language. This approach to distributed programming is powerful, in that it is based based on network locality, and is thus highly responsive to the underlying topology of the system.

This approach is easiest to illustrate with the simple example of constructing a spanning tree. Take a distributed system with the following state variables:

id: A unique integer identifier for each process.

isRoot: A boolean flag identifying the root of the spanning tree.

parent: An integer variable, initialized to the constant *nil*.

To create a spanning tree rooted at the process marked by *isRoot*, we use following predicate/action pairs (where predicate and action are separated by $\rightarrow$):

$$Q_1(a_1, a_2) : isRoot_{a_1} \rightarrow parent_{a_1} = id_{a_1}$$

$$Q_2(a_1, a_2) : (parent_{a_1} \neq nil) \wedge (parent_{a_2} = nil) \rightarrow parent_{a_2} = id_{a_1}$$

Predicate Q_1 identifies the root of the tree, and sets the root's *parent* to its *id*. Predicate Q_2 finds pairs of processes where process a_1 is a member of the spanning tree, while process a_2 is not. Process a_2 then sets its *parent* variable to point to a_1. These two predicate/action pairs, when executed by every process, form a global spanning tree in the system. Importantly, they do so without any explicitly specified messaging, and without requiring any multi-hop communication. We discuss this topic at length in our work on declarative programming for modular robots [De Rosa et al. 2008; Ashley-Rollman et al. 2007].

9. CONCLUSIONS

In this paper, we have formally introduced the concept of locally distributed predicates, and shown two algorithms for detecting them in FIFO systems. The first is a consistent cut protocol, capable of detecting strong stable properties, while second is an inhibitory algorithm that creates distributed snapshots. We have shown that, although their complexity in fully-connected networks is inferior to that of existing algorithms, the message- and state-complexity of the algorithms is scale-free in constant-degree graphs. Finally, we have demonstrated the utility of LDPs as the foundation for programming and debugging tools in modular robotics and sensor networks.

While LDPs represent a first step in developing topology-aware predicates, there are numerous opportunities to improve and refine them. The most obvious lack in

LDPs is the inability to describe topologies other than linear chains. The ability to represent trees, DAGs, or arbitrary subgraphs in predicates would allow the expression of a much larger set of properties. The detection algorithms for LDPs can also be improved, as they are relatively expensive. Finally, the creation of new algorithms for detecting non-stable LDPs, or for performing detection over non-FIFO channels, would increase the set of applications for which LDPs are appropriate.

ACKNOWLEDGMENTS

This work was partially funded by the National Science Foundation (NSF) under grant no. CNS-0428738, Intel Labs Pittsburgh, Carnegie Mellon University, and a generous contribution from the Microsoft Corporation. Special thanks to the members of the Claytronics research group. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

REFERENCES

Ashley-Rollman, M., De Rosa, M., Srinivasa, S., Pillai, P., Goldstein, S., and Campbell, J. 2007. Declarative programming for modular robots. In *IROS 2007 Workshop on Modular Robots*.

Babao, Ö., Marzullo, K., and Marzullo, K. 1993. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems*. Addison-wesley, 55–96.

Baldoni, R. and Raynal, M. 2002. Fundamentals of distributed computing: A practical tour of vector clock systems. In *Distributed Systems Online*.

Butler, Z., Kotay, K., Rus, D., and Tomita, K. 2002. Generic decentralized control for a class of self-reconfigurable robots. In *Intl Conf on Robotics and Automation (ICRA)*. IEEE, 809–816.

Chandy, K. M. and Lamport, L. 1985. Distributed snapshots: Determining global states in distributed systems. *ACM Transactions on Computer Systems 3,* 1 (February), 63–75.

Cooper, R. and Marzullo, K. 1991. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*. Vol. 26. 167–174.

Critchlow, C. and Taylor, K. 1990. The inhibition spectrum and the achievement of causal consistency. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*. ACM, New York, NY, USA, 31–42.

De Rosa, M., Goldstein, S. C., Lee, P., Campbell, J., and Pillai, P. 2008. Distributed watchpoints: Debugging large modular robotic systems. *International Journal of Robotics Research 27,* 3 (Special Issue on Modular Robotics).

De Rosa, M., Goldstein, S. C., Lee, P., Pillai, P., and Campbell, J. 2009. A tale of two planners: Modular robotic planning with LDP. In *2009 IEEE/RSJ International Confernce on Intelligent Robots and Systems (IROS)*.

De Rosa, M., Goldstein, S. C., Lee, P., Pillai, P. S., and Campbell, J. 2008. Programming modular robots with locally distributed predicates. In *Proceedings of the IEEE ICRA*.

Fidge, C. 1991. Logical time in distributed computing systems. *Computer 24,* 8 (Aug), 28–33.

Karp, B. and Kung, H. T. 2000. Greedy perimeter state routing for wireless networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*. Boston, Massachusetts, 243–254.

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

Kshemkalyani, A. D., Raynal, M., and Singhal, M. 1995. An introduction to snapshot algorithms in distributed computing. *Distributed Systems Engineering 2,* 4, 224–233.

Lai, T. H. and Yang, T. H. 1987. On distributed snapshots. *Inf. Process. Lett. 25,* 3, 153–158.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7, 558–565.

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. 2004. Tinyos: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag.

Marzullo, K. and Sabel, L. S. 1994. Efficient detection of a class of stable properties. *Distrib. Comput. 8,* 2, 81–91.

Obermarck, R. 1982. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7,* 2, 187–208.

Raynal, M. 1999. Illustrating the use of vector clocks in property detection: An example and a counter-example. In *Euro-Par '99: Proceedings of the 5th International Euro-Par Conference on Parallel Processing*. Springer-Verlag, London, UK, 806–814.

Rus, D. and Chirikjian, G., Eds. 2001. *Special Issue on Self-Reconfiguring Robots*. Vol. 10,1. Autonomous Robotics.

Schiper, A. and Sandoz, A. 1994. Strong stable properties in distributed systems. *Distrib. Comput. 8,* 2, 93–103.

Tel, G. and Mattern, F. 1993. The derivation of distributed termination detection algorithms from garbage collection schemes. In *ACM Transactions on Programming Languages and Systems*. Vol. 15. ACM Press, 1–35.

Venkatesan, S. 1989. Message-optimal incremental snapshots. In *Distributed Computing Systems, 1989., 9th International Conference on*. 53–60.

Received August 2009;