Journal of Parallel and Distributed Computing **62**, 1104–1119 (2002)
doi:10.1006/jpdc.2001.1818

Communication Patterns in Distributed Computations[1]

Ajay D. Kshemkalyani[2,3]

Computer Science Department (mc 152), 851 South Morgan Street, University of Illinois at Chicago, Chicago, Illinois 60607-7053
E-mail: ajayk@cs.uic.edu

and

Mukesh Singhal

Department of Computer Science, University of Kentucky, 301 Rose Street, Lexington, Kentucky 40506
E-mail: singhal@cs.uky.edu

Received April 11, 2001; accepted January 9, 2002

This paper identifies two classes of communication patterns that occur in distributed computations and explores their properties. It first examines local patterns, primarily *IO* and *OI intervals*, that occur at nodes in distributed computations. These local patterns form building blocks that are then used to define the global patterns, termed *segments* and *paths*, that occur across nodes in distributed computations. By controlling the predicates on the local patterns used to define segments and paths, various types of segments and paths can be defined. While a causal chain captures only the causality relation, it turns out that some of the other message sequences that do not capture causality also play a significant role in the analysis of a distributed computation. The paper presents a framework and shows that a number of key concepts and structures characterizing distributed computations are specific instantiations of the communication patterns identified in the framework. © 2002 Elsevier Science (USA)

Key Words: causality; communication; crown; distributed computation; path; segment; zigzag path.

[1]An earlier version of this paper appeared as: A. Kshemkalyani and M. Singhal, Universal constructs in distributed computations, *in* "Proceedings of Euro-Par '99 Parallel Processing, 5th International Euro-Par Conference," Toulouse, France, Lecture Notes in Computer Science 1685, pp. 795–805, Springer, August 1999. (Also appears as Technical Report TR 29.2136, IBM Corporation, March 1996.)

[2]To whom correspondence should be addressed.

[3]This author's work was supported in part by the National Science Foundation under Grant No. CCR-9875617.

0743-7315/02 $35.00

© 2002 Elsevier Science (USA)
All rights reserved.

1. INTRODUCTION

In a distributed computation, processes exchange information via messages. As process execution is typically asynchronous and message delays are unpredictable, it is difficult to predict and control the evolution of a distributed computation. Analyzing the structure of a distributed computation helps to understand the concurrency and leads to a better design of distributed applications, algorithms, and systems. To this end, this paper identifies two classes of communication patterns that occur in every distributed computation and examines their properties. The first class of patterns consists of local patterns or intervals, primarily *IO* and *OI intervals*, that occur at processes [14]. These local patterns are specified in terms of message send and message receive events at a process, and are distinguished by the order in which a pair of messages is sent and/or received by a process. Domain-specific predicates can be defined on how the interval at one process is related to the interval at another process. The use of such predicates on intervals at different processes allows intervals to be used as building blocks to formulate the second class of patterns, which is comprised of two global patterns, termed *segments* and *paths* [14]. These global patterns occur across processes in a distributed computation and signify the flow of information and coupling among the events at different processes. By controlling the predicates on the intervals used to define segments and paths, different types of segments and paths can be defined. Segments and paths generalize causal chains to other message sequences that also play a significant role in the analysis of a distributed computation.

This paper gives a framework in which key concepts and structures characterizing distributed computations in different contexts can be uniformly viewed and understood. The communication patterns identified in this framework are shown to be generalizations of those used in areas or problems such as: formulating the temporal interactions of intervals [13], synchronous and causally ordered communication [9], determining size of logical clocks [8, 20], designing distributed implementations for multilevel secure replicated databases and hierarchically decomposed databases [1, 2], ordering of concurrent events without synchronization [1], transfer of knowledge [7], concurrency measures [10], determining necessary and sufficient conditions for a consistent global state [6, 19] which is useful in checkpointing and recovery [3–5], and defining distributed deadlocks [14]. Thus, the paper shows that key concepts and structures in areas such as the above are instantiations of the communication patterns identified in the presented framework.

Section 2 gives the system model. Section 3 defines the local patterns and examines their properties. Section 4 gives examples of predicates that are used to couple local patterns to form global patterns. Section 5 defines the global patterns that occur across nodes and shows that they are generalizations of key concepts and structures used in the applications listed above. Section 6 concludes.

2. SYSTEM MODEL

The system is a network of N nodes (sites) with a logical channel between each pair of nodes. The nodes communicate by passing messages over the logical channels and

do not share memory. We assume, without loss of generality, that each node in the system has one process running on it. Hence, nodes are synonymous to processes. Process executions and message transfers are asynchronous. Messages are delivered reliably but not necessarily in the order sent.

The system *execution* or *computation* consists of a sequence of events at each node. There are three types of events at a node: message send events, message receive events, and internal events. Let s_i^x and r_j^x denote the send and the receive events at which the message with label x is sent at node i and received at node j, respectively. The superscript and/or subscript will be omitted when it is not important. Let $dest(s_i^x)$ denote the destination of the message sent at s_i^x. A distributed computation associates with each node i a totally ordered set C_i of events. Let $C = \bigcup C_i$ be the possibly infinite set of all events. The state of a node is defined by the values of the variables associated with its computation, which are a function of the history of events executed by it at any time. A distributed computation is represented by the poset $(C, \prec)$, where $\prec$ is the causality relation on C [15].

DEFINITION 1. For any event a and b in C, a happens causally before b, denoted as $a \prec b$, if (i) $a, b \in C_i$ and a occurs before b, or (ii) a is the sending of a message and b is the receipt of the same message, or (iii) there exists some event $c \in C$ such that $a \prec c$ and $c \prec b$.

It is observed that all real computations are acyclic and $(C, \prec)$ is a strict partial order. The global state of the system is the collection of the states of each node after the execution of a left-closed subset of C. (A set C' is left-closed iff for any $a, b \in C$, we have $b \in C' \wedge a \prec b \Rightarrow a \in C'$.)

3. LOCAL COMMUNICATION PATTERNS: INTERVALS AT A NODE

This section formalizes the local communication patterns that occur at nodes as a result of message sends and receives. The following sections show how these patterns are used as building blocks to formulate two global patterns that occur across nodes.

At the time a node i sends a message at s_i, an "outward dependency" gets established at i. At the time a node i receives a message at r_i, an "inward dependency" gets established at i. An *interval* at a node is the period between the times that two such dependencies get established. There are two main types of intervals, shown in Figs. 1a and 1b, based on whether the inward dependency is established before the outward dependency or vice versa. The former interval which is the duration between a receive event and a later send event is an *IO interval*. The latter interval which is the duration between a send event and a later receive event is an *OI interval*. Analogously, *II intervals* and *OO intervals* can also be defined.

The formation of an interval at a node signifies the potential participation of the node in global communication patterns that span across nodes. Note that intervals at a node can overlap. For example, in Fig. 3, the following pairs of intervals overlap at node 3: (i) the OI interval between s_3^3 and r_3^2, the OI interval between s_3^3 and r_3^5; (ii) the

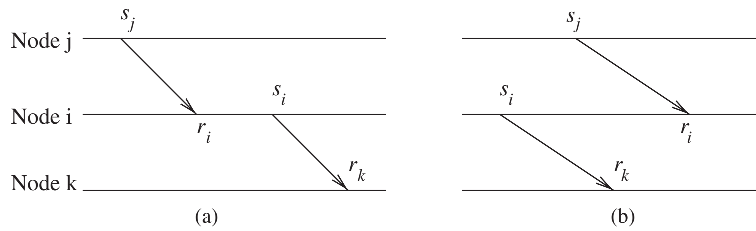

FIG. 1. IO and OI intervals. (a) IO interval at node i. (b) OI interval at node i.

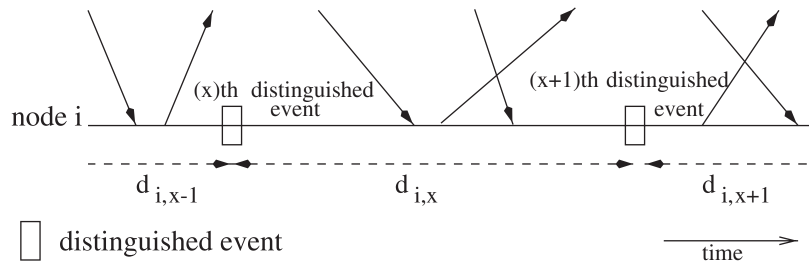

FIG. 2. Distinguished events and durations $d_{i,x}$.

IO interval between r_3^2 and s_3^6, the IO interval between r_3^5 and s_3^6; and (iii) the OI interval between s_3^3 and r_3^5, the IO interval between r_3^2 and s_3^6. If there are k send and receive events at a node, there are $(k(k-1))/2$ intervals at that node.

Each node has a set of application-specific semantically defined "distinguished" events that are identified by monotonically increasing functions such as the sequence number of the events at the node. A distinguished event may be forced to occur by a receive event which is in response to a remote send event, and hence may not be locally deterministic. An example of a distinguished event is an event that takes a checkpoint of the local state of a node. The time span from the xth to the $(x+1)$th distinguished event at node i is called the xth duration at i and is denoted $d_{i,x}$. Also define function $D(e_i) = x$, where e_i occurs in duration $d_{i,x}$. Note that the endpoints of an interval are identified by communication events, whereas the endpoints of a duration $d_{i,x}$ are identified by distinguished events which may or may not be communication events. Figure 2 illustrates this relationship. Intervals of interest to an application are those that satisfy a certain application-specific relationship on the durations in which the send and receive events identifying the interval occur. Each duration $d_{i,x}$ is associated with a predicate $\Phi_{i,x}$ which is true from the start of that duration until some instant within that duration. The predicate can be made false by a receive event in response to a remote send event, and hence may not be under full local control. An example of such a predicate would be "node i has made its xth request and is waiting for a reply". The predicate becomes false when a reply is received. A send event s_i and a receive event r_i can be related at a node i in one of the following ways (and an analogous classification using a pair of

send events or a pair of receive events can be done for OO and II intervals, respectively).

1. $D(s_i) - D(r_i) = 0$: Events s_i and r_i belong to the same duration and identify an IO or an OI interval, based on whether $r_i \prec s_i$, or vice versa, resp.
2. $D(s_i) - D(r_i) > 0$: In this case, events s_i and r_i identify an IO interval.
3. $D(s_i) - D(r_i) < 0$: In this case, events s_i and r_i identify an OI interval.

The notion of a duration as being demarcated by distinguished events is useful to selectively identify (by specifying application-dependent conditions) IO and OI intervals at various nodes, that can potentially be combined to form different types of segments and paths (global patterns).

Section 4 identifies ways in which IO and OI intervals at different nodes can be coupled together. Section 5 defines the global communication patterns using various ways to couple the IO and OI intervals, and shows their applications. Analogously, applications that use II and OO intervals can also be identified. In the remainder of this section, we examine some of the ways in which the send and receive events that identify intervals have been used in related contexts.

In the study of the temporal interactions of intervals in distributed systems, a set of 29 possible orthogonal ways (i.e., exhaustive set of mutually exclusive possibilities) in which two time spans at two different nodes in a distributed computation may be related to each other in terms of causality using the dense time assumption was identified in [13]. Analogously, a set of 40 possible orthogonal ways was identified for the nondense model of time [13]. The analyses leading to these results relied on the existence (or nonexistence) of IO and OI intervals during the two given time spans, and how these IO and OI intervals were related to one another.

The send and receive events, which in pairs identify intervals at a node, have been used as the building blocks of the input/output automata model of asynchronous distributed systems [17]. In this model, each node or system component is viewed as an automaton with a set of states, a set of initial states, a transition relation, and an action signature which contains disjoint sets of input actions, output actions, and internal actions. Automaton A communicates to automaton B when an output action of A is the input action of B; automatas can be composed hierarchically to build a complex system. This input/output automata model has been used widely in the modular design, specification, and verification of distributed algorithms and systems [16].

4. COUPLING LOCAL PATTERNS

Domain-specific predicates can be defined on how an IO or OI interval at one node is related to an IO or OI interval at another node. The use of such predicates on IO and OI intervals at different nodes allows IO and OI intervals to be used as building blocks to formulate the global patterns: segments and paths. These global patterns occur across different nodes and signify a sequence of message exchanges such that any two adjacent messages in the sequence are related at a node by an IO

or an OI interval. By controlling the predicates on the IO and OI intervals used to define segments and paths, different types of segments and paths can be defined.

We introduce some example predicates using which the various global communication patterns in Section 5 are defined. In a computation, let there exist a sequence $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ of send events on nodes $i_j \in \{i_1, i_2, \ldots, i_n\}$ satisfying a combination of the following conditions (henceforth, $i_j \in \{i_1, i_2, \ldots, i_n\}$).

(C1) *Convey predicate to successor*: $D(s_{i_j}) = x_{i_j}$ and $dest(s_{i_j}) = i_{j+1}$, for $1 \leqslant j \leqslant n-1$. The predicate $\Phi_{i_j, x_{i_j}}$ has been conveyed to the (successor) node having the next event in the sequence of send events.

(C2) *Predicate conveyed from predecessor*: A node i_j (except for $j = 1$) has received the message sent by i_{j-1} at $s_{i_{j-1}}$ before s_{i_j}.
A message (potentially containing $\Phi_{i_{j-1}, x_{i_{j-1}}}$) has been received from the (predecessor) node having the previous event in the sequence of send events.

(C3) *No local action that violates predecessor's predicate*: Each node i_j (except for $j = 1$) has not invalidated the predicate $\Phi_{i_{j-1}, x_{i_{j-1}}}$ at node i_{j-1}.
No local action has occurred to invalidate the predicate $\Phi_{i_{j-1}, x_{i_{j-1}}}$ that was true of the predecessor when it sent a message conveying the predicate (as per condition (C1)). Recall from the discussion of predicates that a predicate at i_{j-1} can be made false by events nonlocal to it (e.g., message send event from i_j to i_{j-1} that makes the predicate false when the message is received).

(C4) *No knowledge of violation of predecessor's predicate*: A node i_j (except for $j = 1$) has not received any message, in the causal past of which i_{j-1}'s predicate $\Phi_{i_{j-1}, x_{i_{j-1}}}$ got invalidated.
No message has been received from any node that indicates the predecessor's predicate $\Phi_{i_{j-1}, x_{i_{j-1}}}$ is no longer valid. Refer to the discussion under (C3) which discusses how $\Phi_{i_{j-1}, x_{i_{j-1}}}$ can get falsified due to events nonlocal to it.

(C5) *Remote predicates observed to be valid in duration*: Each node i_j is in its x_{i_j}th duration and $\Phi_{i_j, x_{i_j}}$ is currently true, as observed nonlocally.
While $\Phi_{i_j, x_{i_j}}$ is defined to be true from the start of the x_{i_j}th duration at i_j until some instant in that duration, this condition is useful for making assertions about $\Phi_{i_j, x_{i_j}}$ and i_j at a different node, or for making assertions about the global state, when the state at i_j is not directly observable by other nodes.

(C6) *Duration containing send event does not occur before duration containing receive event*: $D(s_{i_j}) \geqslant D(r_{i_j})$.
This signifies that at node i_j, there is either an IO interval, or an OI interval in which the send and receive events are in the same duration.

The above six examples of predicates are used to define segments and paths of various types in Section 5. It should be emphasized that other predicates can also be defined, based on the application and context, to define other variants of segments and paths.

5. GLOBAL COMMUNICATION PATTERNS: PATHS AND SEGMENTS

This section defines global patterns that span nodes in a computation. It then shows that several key concepts and structures characterizing distributed

computations are instantiations of and can be expressed using these patterns. Specifically, the following three versions of segments and paths are presented based on the semantics attached to the events identifying IO and OI intervals.

- The first version (Section 5.1) is for a *general* computation where no restrictions are imposed and any s_i and any r_i events at a node i can be used to identify OI and IO intervals. We show its usage in characterizing distributed computations by identifying structures like a crown which are used in a wide range of results, deriving concurrency measures, and analyzing knowledge transfer.
- In the second version (Section 5.2), distinguished events are assigned values of a *monotonically* nondecreasing function. We show its usage in characterizing global checkpoints.
- In the third version (Section 5.3), the distinguished events signify participation in a stable property. We show its usage in characterizing *stable* properties like distributed deadlocks.

Each of the three versions of segment and path defined will be subscripted by g, m, and s, respectively.

5.1. Segments and Paths for General Computations

In a general computation, no semantics is attached to the events identifying an interval and no constraints are imposed on the relation between $D(s_i)$ and $D(r_i)$. Thus, all s_i and r_i events at a node i are considered in identifying intervals.

Definition 2. A "*segment*" for a general computation, denoted $S_g(s_{i_1}, r_{i_{n+1}})$, is a sequence of events $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ satisfying (C1) $\wedge$ (C2).

Every event in a segment occurs at a node that has sent a message to the node at which the successor event in the segment occurs. (Henceforth, a reference to "a node on a segment/path" will mean "a node with an event on a segment/path".) Moreover, when a node i_j sends the message at s_{i_j} (as per (C1)), the message sent at the previous event $s_{i_{j-1}}$ in the sequence has been received (as per (C2)). Therefore, a segment denotes a sequence of nodes such that the dependencies on their successor nodes in the segment are created sequentially. That is, $(\forall i_j\colon 1 \leqslant j < n)$, $s_{i_j} \prec s_{i_{j+1}}$. A segment S_g thus represents the widely used concept of the causal chain of messages, in which the events signify completed IO intervals. Other variants of segments defined in later sections are causal chains satisfying additional properties.

For a sequence of events $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ such that $dest(s_{i_j}) = i_{j+1}$ for $1 \leqslant j \leqslant n-1$, it may happen that $\exists j\colon s_{i_j} \not\prec s_{i_{j+1}}$, that is, node i_{j+1} has an OI interval. A *path* is defined next to capture such a sequence of events.

Definition 3. A "*path*" for a general computation, denoted $P_g(s_{i_1}, r_{i_{n+1}})$, is a sequence of events $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ satisfying (C1).

The formation of an interval at a node signifies the participation of the node in a path or a segment. In a segment and in a path, the send events in the sequence

identify a sequence of messages. Thus, a segment or a path implicitly identifies the alternating send and receive events associated with these messages, from the send of the first message to the receive of the last message. In both, adjacent messages must have events at a common node in order to be related by an IO or an OI interval. When following the sequence of events in a path, one can move forward or backward along the timeline at that node. When following the sequence of events in a segment, one must only move forward. Thus, in a path, successive messages are related by either an IO or an OI interval but in a segment, successive messages are related only by IO intervals. Thus, the successive events in a sequence at which outward dependencies are established satisfy a weaker causal relationship in a path than in a segment. Note that a path may contain several segments; a segment is always a path.

Figure 3 gives examples of paths and segments. Some segments are: $\langle s_1^1, s_2^4, s_4^5, s_3^6 \rangle$, $\langle s_2^2, s_3^6 \rangle$, $\langle s_3^3, s_4^5, s_3^6 \rangle$, and all subsequences of the above. By definition, each segment is a path. The following are some paths with at least one OI interval: $\langle s_1^1, s_2^2, s_3^3 \rangle$, $\langle s_1^1, s_2^2, s_3^6 \rangle$, $\langle s_2^2, s_3^3, s_4^5, s_3^6 \rangle$. Subsequences of these are also paths.

Concatenation, prefix, suffix, and other standard string/sequence operations can be formally defined on paths and segments. For example, two paths $P_g(s_{i_1}, r_{i_{m+1}})$ and $P_g(s_{j_1}, r_{j_{n+1}})$ of length m and n, respectively, can be concatenated if $(i_{m+1} = j_1)$ or $(j_{n+1} = i_1)$.

A *maximal path* is an acyclic path which cannot be extended by the addition of a send event at either end. For example, in Fig. 3, $\langle s_1^1, s_2^2, s_3^3, s_4^5, s_3^6 \rangle$ is a maximal path consisting of messages $m1$, $m2$, $m3$, $m5$, and $m6$. The longest maximal path in the computation is $\langle s_2^2, s_3^3, s_4^5, s_3^6, s_1^1, s_2^4 \rangle$ which happens to consist of all the messages in the computation. A *maximal segment* is defined likewise. In Fig. 3, $\langle s_1^1, s_2^4, s_4^5, s_3^6 \rangle$ is a maximal segment consisting of messages $m1$, $m4$, $m5$, and $m6$.

Maximal segments and maximal paths are useful concepts in analyzing properties of a distributed computation. A maximal segment is a causal chain that signifies the maximum length of the serial execution of "thread of control" represented by the segment. On the other hand, a maximal path whose events are related by OI intervals at nodes provides a measure of the concurrency in a computation. The higher the number of OI intervals, the higher the concurrency. The ratio of the average of the sizes of maximal paths to the average of the sizes of maximal segments in a distributed computation is a good indicator of the concurrency in the computation [10].

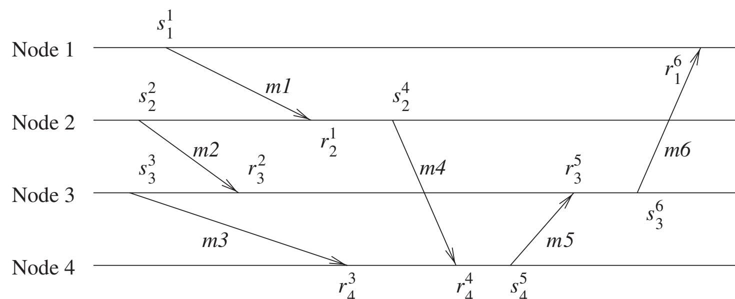

FIG. 3. An example computation.

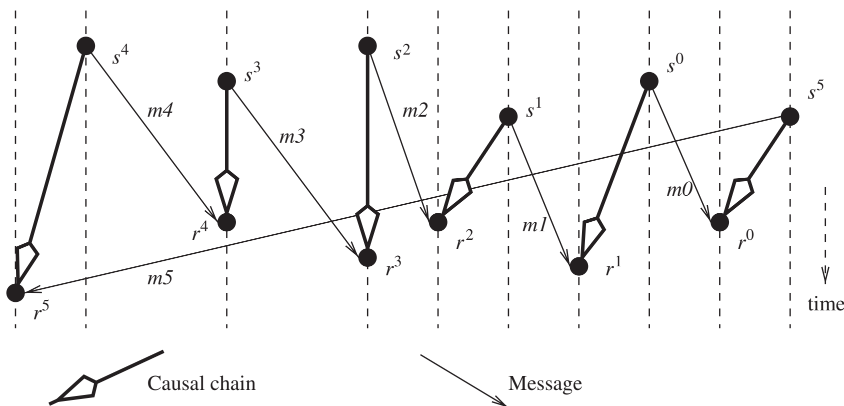

FIG. 4. A crown of size 6.

We next show how segments and paths can be used to express some communication patterns that are important in analyzing distributed computations.

5.1.1. The crown criterion. A crown in a partial order is a specific suborder [23] that has many applications. We first give a definition of a crown tailored to a distributed computation, then describe some of the application areas and results that illustrate the importance of the crown, and then show that the crown is an instantiation of the communication patterns—segments and paths—within the framework of this paper.

DEFINITION 4. Let C be a computation. A crown of size k in C is a sequence $\langle (s^i, r^i),\ i \in \{0, \ldots, k-1\} \rangle$ of pairs of corresponding send and receive events such that: $s^0 \prec r^1,\ s^1 \prec r^2, \ldots, s^{k-2} \prec r^{k-1},\ s^{k-1} \prec r^0$.

Figure 4 shows a crown having six pairs of corresponding send and receive events $(s^i, r^i),\ i \in [0, 5]$. There is also a causal chain (segment S_g) from s^i to $r^{(i+1) \bmod 6}$, for $i \in [0, 5]$.

In a classification of a hierarchy of communication patterns, Charron-Bost *et al.* observed that a distributed algorithm designed to run correctly on asynchronous systems (called *A-computations*) may not run correctly on synchronous systems—an algorithm that runs on an asynchronous system may *deadlock* on a synchronous system [9]. A-computations that can be realized under synchronous communication are called *realizable with synchronous communication* (RSC) computations. Formally, a computation C is RSC if there exists a nonseparated linear extension of the poset $(C, \prec)$.[4] Charron-Bost *et al.* [9] showed that RSC computations are a proper subset of causally ordered computations, which are a proper subset of FIFO computations. Charron-Bost *et al.* [9] developed a criterion (called the *crown criterion*) and showed that an A-computation is RSC, i.e., it can be realized on a system with synchronous communication, iff it contains no crown.

[4] A nonseparated linear extension of $(C, \prec)$ is a linear extension of $(C, \prec)$ such that for each pair of send event s and corresponding receive event r, the interval $\{x \in C \mid s \prec x \prec r\}$ is empty.

The ordering of events in a distributed computation is a very fundamental problem [15]. For concurrent events, some ordering decisions (such as tie-breaking using the node_id or using a central arbiter node) are inherently artificial and no algorithm suits all applications. For several applications, allowing event ordering (of concurrent events) upon receipt of a message at a given node is desirable [1]. For example, in a transaction processing application, when a node receives a message about a transaction that a timestamping algorithm determines to have been sent in the far past at the receiving node, then the node may be obliged to undo all later transactions. It would have been more efficient for the ordering of that transaction to be determined, at least in part, by the recipient's clock. However, allowing ordering of events upon the receipt of the corresponding message can obviously lead to inconsistent orderings by multiple nodes. Ammann *et al.* addressed the problem "What is the largest class of communication structures that guarantee that local ordering decisions are globally consistent without any further global synchronization?" They answered by identifying crown-freedom in the computation as the necessary and sufficient condition [1]. They then showed that for multilevel, secure replicated databases and for hierarchically decomposed databases, if the communication structure is crown-free, then and only then is a distributed implementation that guarantees globally serializable transaction histories possible (without additional synchronization information) [1, 2].

It was shown by Charron-Bost [8] based on a result by Ore [20] that to capture causality in a distributed computation $(C, \prec)$, i.e., to test $e \prec f$ iff $T(e) < T(f)$, a vector clock of size equal to the dimension of that partial order is necessary and sufficient. The dimension of the partial order can be as large as its width (i.e., number of nodes) as demonstrated by the standard crown S_n^0 [23]. This result based on the crown S_n^0 has wide implications because clocks of size $|N|$ are necessary to capture causality as required for a large range of distributed applications [18].

Definition 4 of a crown specifies the constraints between s^i and $r^{(i+1) \bmod k}$ for $i \in [0, k-1]$. Each such constraint simply represents a segment $S_g(s^i, r^{(i+1) \bmod k})$. Towards our objective, we first show that crowns are equivalently defined in terms of segments, and then recast that definition using a fewer number of segments and paths. This shows that the crown is an instantiation of the communication patterns in our framework.

Definition 5. In terms of segments, a crown of size k in a computation is a sequence $\langle (s^i, r^i),\ i \in \{0, \ldots, k-1\} \rangle$ of pairs of corresponding send and receive events such that $\forall i \in [0, k-1],\ S_g(s^i, r^{(i+1) \bmod k})$.

To simplify notation, the crown will also be expressed as $\{S_g(s^i, r^{(i+1) \bmod k}):\ i \in [0, k-1]\}$.

Example 1. The crown in Fig. 4 is expressed using segments as: $CROWN = \{S_g(s^i, r^{(i+1) \bmod k}) : i \in [0, 5]\}$.

Refinement of Definition 5: Definition 5 expresses a crown of size k in terms of k segments. A crown of size k can generally be expressed in terms of fewer than k segments and paths.

A segment $S_g(s^i, r^j)$ such that events s^i and r^j lie on the same node is called a *local* segment. Note that in Fig. 4, (i) segments $S_g(s^2, r^3)$ and $S_g(s^3, r^4)$ are local segments and (ii) these two segments are connected by message (s^3, r^3). In this situation, segments $S_g(s^2, r^3)$ and $S_g(s^3, r^4)$ can be represented by path $\langle s^4, s^3, s^2 \rangle$. Consequently, the conditions represented by segments $S_g(s^2, r^3)$ and $S_g(s^3, r^4)$ in the expression of the crown can be equivalently stated in terms of path $\langle s^4, s^3, s^2 \rangle$, which happens to contain only OI intervals. Given a crown, we present an algorithm that replaces clusters of local segments connected by messages, by equivalent paths. This algorithm compacts consecutive segments into paths wherever possible—such paths consist of OI intervals only. (Note that a cyclic path with OI intervals only is always a crown.)

A Crown-Compaction Algorithm

1. $CR_ALT = CROWN$.
2. Identify each maximal sequence of consecutive integers, modulo k, from x to y satisfying $\forall j \in [x, (y) \bmod k]$, s^j and $r^{(j+1) \bmod k}$ occur on the same node. For each such sequence, do the following.
   (a) $CR_ALT = CR_ALT \setminus \{S_g(s^i, r^{(i+1) \bmod k})\colon\ i \in [x, (y) \bmod k]\}$.
   (b) $CR_ALT = CR_ALT \cup \{\langle s^{(y+1) \bmod k}, s^y, s^{(y-1) \bmod k}, \ldots, s^{(x+1) \bmod k}, s^x \rangle\}$.

EXAMPLE 1 (*Contd.*). In the crown in Fig. 4, s^2 and r^3 lie on the same node, and s^3 and r^4 lie on the same node. As there is a range of consecutive integers $[x, y] = [2, 3]$ such that $\forall i \in [2, 3]$, s^i and $r^{(i+1) \bmod k}$ lie on the same node, segments $S_g(s^2, r^3)$ and $S_g(s^3, r^4)$ can be replaced by path $\langle s^4, s^3, s^2 \rangle$, Hence, $CR_ALT = \{S_g(s^i, r^{(i+1) \bmod k})\colon\ i \in \{0, 1, 4, 5\}\} \cup \{\langle s^4, s^3, s^2 \rangle\}$.

Thus, a crown which is an example of various structures in distributed computations is an instantiation of the paths and segments of the proposed framework.

5.1.2. Knowledge transfer. Knowledge in distributed systems is about local and global facts defined on the states of the nodes. These facts are defined as temporal and spatial predicates over the variables of the nodes. Knowledge plays a significant role in the evaluation of global predicates, debugging, monitoring, establishing breakpoints, evaluating triggers, industrial process control, and controlling a distributed execution [22].

Knowledge is transferred among nodes through send and receive events [7]; the extent of knowledge dissemination is determined by the message communication pattern among nodes, which is defined by the causality relation between events. A segment from event e_i at a node i to event e_j at a node j signifies the flow of knowledge of node i's state preceding event e_i to all the local states at node j, following event e_j. For example, in Fig. 3, messages $m1, m4$, and $m5$ constitute a segment $\langle s_1^1, s_2^4, s_4^5 \rangle$, and these messages transfer the knowledge about the local state of node 1 just before event s_1^1 to event r_3^5. A path that has an OI interval denotes a disrupted transfer of knowledge among the nodes along the path. In Fig. 3, knowledge about the local state of node 1 just before event s_1^1 is not transferred to

event r_3^2. The knowledge transfer is disrupted at node 2 due to the OI interval formed by s_2^2 and r_2^1. Thus, the paths and segments of the framework are useful tools that have been used to identify the extent of knowledge transfer.

5.2. *Segments and Paths for Monotonically Nondecreasing Functions*

In distributed computations with monotonically nondecreasing functions at nodes, the distinguished events at a node are associated with monotonically nondecreasing values. An example of such nondecreasing values is the local clock time at the occurrence of an event at a node [15].

The definition of a segment for a monotonically nondecreasing function (Definition 6) is the same as for a general function (Definition 2). The definition of a path for a monotonically nondecreasing function (Definition 7) differs from the corresponding Definition 3 in that the events of an OI interval must belong to the same duration.

DEFINITION 6. A "*segment*" for a monotonically nondecreasing function, denoted $S_m(s_{i_1}, r_{i_{n+1}})$, is a sequence of events $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ satisfying (C1) $\wedge$ (C2).

DEFINITION 7. A "*path*" for a monotonically nondecreasing function, denoted $P_m(s_{i_1}, r_{i_{n+1}})$, is a sequence of events $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ satisfying (C1) $\wedge$ (C6).

A *closed path* for a monotonically nondecreasing function is a path $P_m(s_{i_1}, r_{i_{n+1}})$, such that events s_{i_1} and $r_{i_{n+1}}$ occur at the same node (i.e., $i_1 = i_{n+1}$).

We now show an application (checkpointing) that uses instantiations of segments and paths in computations with monotonically nondecreasing functions.

5.2.1. Necessary and sufficient conditions for a global snapshot: zigzag paths. Checkpointing is used in fault-tolerant computing [4, 5], and parallel and distributed debugging [22]. Each node can take local checkpoints asynchronously; a consistent global checkpoint is constructed by choosing a local checkpoint from each node. Checkpoints are the "distinguished events" which demarcate consecutive durations at nodes. The xth duration (or xth *checkpoint interval*) at a node denotes the computation from its xth to its $(x+1)$th checkpoint.

An important problem is to determine if an arbitrary set of local checkpoints belongs to a consistent global checkpoint [6]. Netzer and Xu used the zigzag path, a generalization of Lamport's causality relation [15], and showed that two local checkpoints cannot lie on a consistent global checkpoint iff a zigzag path exists between the checkpoints [19]. A zigzag path is defined next. Let $C_{i,x}$ denote the xth local checkpoint at node i and let $e_{i,x}$ denote the event of taking $C_{i,x}$.

DEFINITION 8. A zigzag path exists from $C_{i,x}$ to $C_{j,y}$ iff there are messages $m_1, m_2, \ldots, m_n$ $(n > 1)$ such that

1. m_1 is sent by node i after $C_{i,x}$;
2. if m_k $(1 \leqslant k < n)$ is received at node r, then m_{k+1} is sent by r in the same or a later checkpoint interval;
3. m_n is received by process j before $C_{j,y}$.

In Fig. 5, messages m_1, m_2, and m_3 form a zigzag path from checkpoint C_{11} at node 1 to checkpoint C_{32} at node 3. Likewise, messages $m4$, $m5$, and $m6$ form a zigzag path from checkpoint C_{12} at node 1 to checkpoint C_{42} at node 4. Note from Definition 8 that a zigzag path is a chain of messages that are connected by OI or IO intervals at nodes. Thus, a zigzag path is nothing but a "path" (Definition 7) and can be expressed using paths as follows:

DEFINITION 9. A zigzag path exists from $C_{i,x}$ to $C_{j,y}$ iff there exists a path $P_m = \langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ such that (i) $e_{i,x} \prec s_{i_1}$, and (ii) a message sent at s_{i_n} to j is received before $e_{j,y}$.

A checkpoint is defined to be on a Z-cycle iff there is a zigzag path from the checkpoint to itself [19]. In Fig. 5, checkpoint C_{32} lies on a Z-cycle consisting of messages $m6$ and $m3$. Observe that a Z-cycle is nothing but the closed path of Definition 7 and hence an instantiation of paths in the presented framework.

It was shown in [19] that a checkpoint can be part of a consistent snapshot iff it is not involved in a Z-cycle. Based on this result, researchers further optimized the number of checkpoints taken asynchronously so that each checkpoint was guaranteed to be part of some consistent snapshot (thus it was guaranteed not to be a part of a Z-cycle or a closed path), and was thus not wasted [3, 11].

5.3. Segments and Paths for Stable Properties

A stable property is a property of the system state such that once it becomes true, it continues to hold unless there is external intervention [21]. Examples of such properties are deadlocks, termination of a computation, etc. In this section, segments and paths are defined for the stable property of deadlocks [12, 14]. We believe that a similar approach with appropriate modifications can be used for other stable properties.

5.3.1. Conditions for deadlocks. We consider deadlocks in the request–reply model. In this model, a process sends a request and blocks until it receives a reply to its request. "Distinguished" events at a node are the events at which a node sends a request and blocks waiting for a reply. The predicate $\Phi_{i,x}$ stands for "node i is blocked on the request it sent at its xth distinguished event". This predicate becomes true at the start of the duration between two distinguished events and becomes false

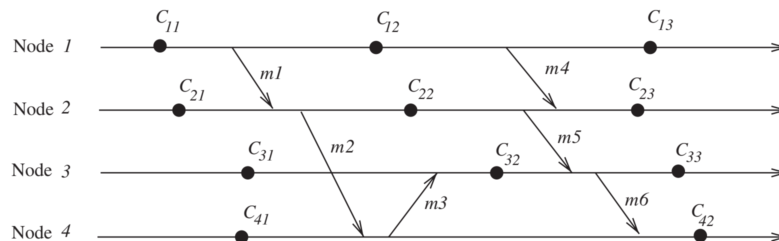

FIG. 5. Zigzag paths.

on the receipt of the reply at some time before the next distinguished event. In this context, a segment and a path are defined next [14].

DEFINITION 10. A "*segment*" in the request–reply model, denoted $S_s(s_{i_1}, r_{i_{n+1}})$, is a sequence of events $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ satisfying the following conditions:

(I) (C1) $\wedge$ (C2) $\wedge$ (C3) $\wedge$ (C4). /* conditions on distinguished events. */
(II) (C5). /* conditions when the system is observed. */

DEFINITION 11. A "*path*" in the request–reply model, denoted $P_s(s_{i_1}, r_{i_{n+1}})$, is a sequence of events $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle$ satisfying the following conditions:

(I) (C1) $\wedge$ ((C2) $\Rightarrow$ ((C3) $\wedge$ (C4))). /* conditions on distinguished events. */
(II) (C5). /* conditions when the system is observed. */

Condition (C2) indicates that the request sent at $s_{i_{j-1}}$ has been received before s_{i_j}. Condition (C3) indicates that i_j has not sent back a reply to i_{j-1} and thus has not invalidated $\Phi_{i_{j-1},x_{i_{j-1}}}$. By condition (C4), i_j has not received a message indicating that i_{j-1} got unblocked, i.e., $\Phi_{i_{j-1},x_{i_{j-1}}}$ got invalidated. In Definition 10, condition (C1) $\wedge$ (C2) $\wedge$ (C3) $\wedge$ (C4) implies that at s_{i_j}, (i) i_j has already received a request sent at $s_{i_{j-1}}$ and (ii) based on its complete causal past, i_j knows that i_{j-1} is blocked at $s_{i_{j-1}}$ on i_j. Thus, in a segment, all events preceding s_{i_j} have sent requests that are directly or transitively blocked on i_j and at each node with an event on the segment, there is an IO interval. In Definition 11, conditions (C1) and ((C2) $\Rightarrow$ ((C3) $\wedge$ (C4))) state that (i) each s_{i_j} has sent a request ((Cl) holds), and (ii) at s_{i_j}, if i_j has already received a request sent at $s_{i_{j-1}}$ (condition (C2)), then based on its causal past, i_j knows that i_{j-1} is blocked at $s_{i_{j-1}}$ on i_j (from (C3) and (C4)). If condition (C2) is false at node i_j, then the incoming request from node i_{j-1} arrives at node i_j after i_j has sent its request, resulting in an OI interval at i_j. Due to the request–reply model, all nodes on the path will remain blocked forever unless (a) either the last node on the path receives a reply from its successor or (b) some node on the path becomes active by aborting or rolling back or withdrawing its request. As no node in a distributed system has instantaneous knowledge of the entire system, while declaring a segment/path, it must be ensured that the nodes that are believed to be blocked (e.g., expressed in "(Cl) and ((C2) $\Rightarrow$ ((C3) $\wedge$ (C4)))" for a path) are still blocked as per the knowledge of the causal past. Condition (C5) asserts that all nodes on the path are still blocked. A detailed explanation of Definitions 10 and 11 is given in [14].

Paths in which each node is blocked waiting for a reply from its successor and the last node never receives a reply denote deadlocks.

DEFINITION 12. A *closed path* is a path $P_s(s_{i_1}, r_{i_{n+1}})$ such that events s_{i_1} and $r_{i_{n+1}}$ occur at the same node (i.e., $i_1 = i_{n+1}$).

A closed path denotes a deadlock because no node with an event on the closed path will ever receive a reply and get unblocked. A closed path has at least one OI interval. Condition (C5) helps to ensure that false deadlocks are not detected. Thus, a cycle in a Wait-For Graph, which is the condition for deadlock, is a specific instantiation of the paths presented in the framework.

6. CONCLUSION

The paper identified two classes of communication patterns in distributed computations. IO, OI, II, and OO intervals are local patterns that occur at nodes, whereas paths and segments are global patterns which occur across nodes in a distributed computation and are defined in terms of the local patterns. These global patterns signify the flow of information and the type of coupling among the events on nodes. Traditionally, causal chains in a distributed computation have been emphasized in the analysis of a distributed computation. However, as argued in this paper, certain other message sequences that do not capture causality or do so with added semantics play a significant role in the analysis and characterization of distributed computations. We showed that a number of key concepts and structures characterizing distributed computations are instantiations of the proposed patterns and can be expressed using these patterns. By controlling the predicates on the local patterns used to define segments and paths, different types of segments and paths can be defined to address the needs of various applications. As a result, various communication patterns that are relevant in different contexts and for various applications can be represented in a unifying framework. Properties of specific communication patterns studied in the context of one application area can potentially be used for other application areas.

REFERENCES

1. P. Ammann, S. Jajodia, and P. Frankl, Globally consistent event ordering in one-directional distributed environments, *IEEE Trans. Parallel Distrib. Systems* **7**(6) (June 1996), 665–670.
2. P. Ammann and S. Jajodia, Planar lattice security structures for multi-level replicated databases, *in* "Database Security VII: Status and Prospects" (T. Keefe and C. Landwehr, Eds.), pp. 125–134, North-Holland, Amsterdam, 1994.
3. R. Baldoni, J.-M. Helary, and M. Raynal, Rollback-dependency trackability: visible characterizations, *in* "Proceedings of the 18th ACM Symposium on Principles of Distributed Computing, Atlanta, GA," pp. 33–42, 1999.
4. B. Bhargava and P. Leu, Concurrent robust checkpointing and recovery in distributed systems, *in* "Proceedings of the 4th IEEE Conference on Data Engineering, Los Angeles, CA," pp. 154–163, February 1988.
5. B. Bhargava and S. R. Lian, Independent checkpointing and concurrent rollback for recovery in distributed systems—An optimistic approach, *in* "Proceedings of the 7th IEEE Symposium on Reliable Distributed Systems, Columbus, OH," pp. 3–12, October 1988.
6. K. M. Chandy and L. Lamport, Distributed snapshots: determining global states of distributed systems, *ACM Trans. Comput. Systems* **3**(1) (1985), 63–75.
7. K. M. Chandy and J. Misra, How processes learn, *Distrib. Comput.* **1** (1986), 40–52.
8. B. Charron-Bost, Concerning the size of logical clocks in distributed systems, *Inform. Process. Lett.* **39**(1) (1991), 11–16.
9. B. Charron-Bost, F. Mattern, and G. Tel, Synchronous, asynchronous, and causally ordered communication, *Distrib. Comput.* **9**(4) (1996), 173–191.
10. C. J. Fidge, A simple run-time concurrency measure, *in* "The Transputer in Australasia (ATOUG-3)" (T. Bossomaier, T. Hintz, and J. Hulskamp, Eds.), pp. 92–41, IOS Press, Amsterdam, 1990.
11. J.-M. Helary, R. Netzer, and M. Raynal, Consistency issues in distributed checkpoints, *IEEE Trans. Software Eng.* **25**(2) (1999), 274–281.

12. A. D. Kshemkalyani and M. Singhal, Correct two-phase and one-phase deadlock detection algorithms for distributed systems, *in* "Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas," pp. 126–129, December, 1990.
13. A. D. Kshemkalyani, Temporal interactions of intervals in distributed systems, *J. Comput. System Sci.* **52**(2) (April 1996), 287–298.
14. A. D. Kshemkalyani and M. Singhal, On characterization and correctness of distributed deadlock detection, *J. Parallel Distrib. Comput.* **22**(1) (July 1994), 44–59. (Also appears as Technical Report TR-06/90-TR15, Ohio State University, 1990.)
15. L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Comm. ACM* **21**(7) (July 1978), 558–565.
16. N. Lynch, "Distributed Algorithms," Morgan-Kaufmann, Los Altos, CA, 1996.
17. N. Lynch and M. Tuttle, Hierarchical correctness proofs for distributed algorithms, *in* "Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, Vancouver, Canada," pp. 137–151, August, 1987.
18. F. Mattern, Virtual time and global states of distributed systems, *in* "Parallel and Distributed Algorithms," (M. Cosnard and P. Quinton Eds.), North-Holland, Amsterdam, pp. 215–226, 1989.
19. R. Netzer and J. Xu, Necessary and sufficient conditions for consistent global snapshots, *IEEE Trans. Parallel Distrib. Systems* **6**(2) (1995), 165–169.
20. O. Ore, "Theory of Graphs", Vol. 38, American Mathematical Society Colloq. Publications, Providence, RI, 1962.
21. A. Schiper and A. Sandoz, Strong stable properties in distributed systems, *Distrib. Comput.* **8** (1994), 93–103.
22. M. Spezialetti and R. Gupta, Debugging distributed programs through the detection of simultaneous events, *in* "Proceedings of the 14th IEEE International Conference on Distributed Computing Systems, Poznan, Poland," pp. 634–641, June 1994.
23. W. Trotter, "Combinatorics and Partially Ordered Sets," The Johns Hopkins University Press, Baltimore, 1992.

AJAY KSHEMKALYANI is an associate professor of computer science at the University of Illinois at Chicago since 2000. He previously spent several years at IBM Research Triangle Park working on various aspects of computer networks. Ajay Kshemkalyani received the Ph.D. and the M.S. in computer and information science from The Ohio State University in 1991 and 1988, respectively, and a B.Tech. in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987. His current research interests include computer networks, distributed computing, algorithms, and concurrent systems. He is a recipient of the National Science Foundation's CAREER award. He is a member of the ACM and a senior member of the IEEE.

MUKESH SINGHAL is a full professor and Gartener Group Endowed Chair in network engineering in the Department of Computer Science at The University of Kentucky, Lexington. He received a Bachelor of Engineering degree in electronics and communication engineering with high distinction from Indian Institute of Technology, Roorkee, India, in 1980 and a Ph.D. in Computer Science from University of Maryland, College Park, in May 1986. His current research interests include distributed systems, mobile computing, computer networks, and computer security. He has published over 145 refereed articles in these areas. He has coauthored three books titled "Data and Computer Communications: Networking and Inter-networking," CRC Press, 2001, "Advanced Concepts in Operating Systems," McGraw–Hill, New York, 1994, and "Readings in Distributed Computing Systems," IEEE Computer Society Press, 1993. He is a fellow of IEEE. He is currently serving on the editorial board of "IEEE Trans. on Knowledge and Data Engineering" and "Computer Networks." From 1998 to 2001, he served as the program director of the Operating Systems and Compilers program at National Science Foundation.