Fast and Message-Efficient Global Snapshot Algorithms for Large-Scale Distributed Systems

Ajay D. Kshemkalyani, *Senior Member*, *IEEE*

Abstract—Large-scale distributed systems such as supercomputers and peer-to-peer systems typically have a fully connected logical topology over a large number of processors. Existing snapshot algorithms in such systems have high response time and/or require a large number of messages, typically $O(n^2)$, where n is the number of processes. In this paper, we present a suite of two algorithms: *simple_tree*, and *hypercube*, that are both fast and require a small number of messages. This makes the algorithms highly scalable. *Simple_tree* requires $O(n)$ messages and has $O(\log\ n)$ response time. *Hypercube* requires $O(n\ \log\ n)$ messages and has $O(\log\ n)$ response time, in addition to having the property that the roles of all the processes are symmetrical. Process symmetry implies greater potential for balanced workload and congestion-freedom. All the algorithms assume non-FIFO channels.

Index Terms—Distributed system, global state, message passing, distributed snapshot, checkpoint, hypercube, supercomputer, cluster, overlay.

1 Introduction and Problem Definition

Consider a distributed system that is modeled as a directed graph (N, L), where N is the set of processors and L is the set of non-FIFO links connecting the processors in a logical application-layer overlay. Let $n = |N|$. The logical overlay is typically fully connected; hence, the all-to-all logical overlay gives $n(n-1)/2$ logical channels. Typically, the rich interconnectivity of the underlying graph (such as a torus, hypercube, and other regular topologies) allows for multiple logical paths among any pair of processors. Such a logical path can be modeled as a non-FIFO channel in the overlay.

A snapshot of a distributed system represents a consistent global state of the system [13]. A snapshot consists of $\langle \bigcup_i \{LS_i\}, \bigcup_{i,j} \{SC_{i,j}\} \rangle$, where LS_i is the local state of processor P_i and $SC_{i,j}$ is the state of the channel $C_{i,j}$ from processor P_i to processor P_j. In a system with non-FIFO channels, $SC_{i,j} = \{\text{messages sent up to } LS_i\} \backslash \{\text{messages received up to } LS_j\}$. Recording distributed snapshots of an execution is a fundamental problem in asynchronous distributed systems [7], [12], [13], [31], [39], [46], [48], [65], and is used for observing various properties of interest [36].

The seminal algorithm by Chandy and Lamport [13] requires sending a special control message called the marker message on each of the logical channels in the system. In the typical case where there exists a fully connected overlay on the network graph, this amounts to a $O(n^2)$ message overhead. Many variants of the Chandy-Lamport algorithm have been proposed; a survey is given in [35]. However, in the traditional literature [13], [39], [48], [65], the best known bound on the number of messages in a distributed algorithm in systems assuming either FIFO or non-FIFO channels is $O(n^2)$ because a marker is sent on each logical channel.

We identify two kinds of large-scale distributed systems in which global snapshots can be taken.

- Present day supercomputing machines and clusters based on the MIMD architecture have hundreds of thousands of processors; see the Top-500 list of supercomputers [66]. Examples of such machines include the BlueGene supercomputer. Such machines are distributed systems as they are often used for solving complex tasks and communicate by message passing. Typically, a single long-lived task is distributed in a supervised manner across the processors which are under a single administrative domain. Coordinated checkpointing (or recording global snapshots) is an important problem in such systems [2], [10], [11], [14], [23], [34], [53], [61].
- Over the past decade, peer-to-peer (P2P) systems have been designed to accommodate hundreds of thousands of processors/end-users communicating by asynchronous message passing over the inter-domain Internet. P2P networks have a self-organizing nature that can evenly share the network load even in the presence of noncooperative peers. The main design principles are driven by scalable object storage and search in a completely autonomous environment, typically characterized by high churn rates and failure rates. Typical interactions are client-server; hence, the notion of coordinated checkpointing appears to be in contradiction to the P2P philosophy by which each peer is free to do as it pleases and can join and leave the system at will. However, recently, it has been shown that P2P technologies are being successfully adopted in large-scale IT enterprise infrastructures to improve the degree of autonomic behavior, and hence, decrease the complexity of the management and the cost of ownership [6], [9], [16], [63], [64]. The enterprise

• *The author is with the Computer Science Department, University of Illinois at Chicago, 851 S. Morgan Street, 1120 Science and Engineering Offices, Chicago, IL 60607. E-mail: ajay@uic.edu.*

Manuscript received 20 Aug. 2009; revised 15 Nov. 2009; accepted 13 Jan. 2010; published online 25 Jan. 2010.
Recommended for acceptance by M. Raynal.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-08-0379.
Digital Object Identifier no. 10.1109/TPDS.2010.24.

setting is very different from the interdomain one: peers in the enterprise infrastructure work cooperatively in a managed environment, and thus, they are much more stable than those in Internet-based applications. Operations that have to be executed on the peers by enterprise applications are more complex than the simple search and store operations implemented by Internet-based applications. For example, a peer in an enterprise setting may be required to take control of the P2P system for auditing, monitoring, or maintenance procedures. For such auditing, monitoring, and maintenance procedures, it becomes necessary to perform coordinated checkpointing and snapshot the system state.

Ni et al. [52] have developed a middleware called P2P Distributed Virtual Machine (P2P-DVM) to allow scientists' resources to be readily shared over the Internet in a P2P manner. Along with MPI-based message passing over P2P networks, P2P-DVM also provides decentralized coordinated checkpointing and restart functionality. Further, emerging P2P applications such as Massive Multiuser Virtual Environments (MMVEs) and Massive Multiuser Online Gaming (MMOG) [21], [27] deal with a unified application context across the peers, and hence, there arises the need for consistently observing the system state and coordinated checkpointing.

A message overhead of $O(n^2)$ messages per snapshot becomes too expensive and is not scalable as the number of processors increases. Recent work has focused on reducing the snapshot complexity in such large-scale systems [23], [24].

In this paper, we present a suite of two algorithms: *simple_tree* and *hypercube*, that are both fast and require a small number of messages. This makes the algorithms highly scalable. *Simple_tree* requires $O(n)$ messages and has $O(\log\ n)$ response time. *Hypercube* requires $O(n\ \log\ n)$ messages and has $O(\log\ n)$ response time, in addition to having the property that the roles of all the processors are symmetrical. Processor symmetry implies greater potential for balanced workload and congestion-freedom. Each processor sends $\log\ n$ messages. The sizes of the messages form a geometric series, and the sum of the message sizes sent by any processor is $O(n)$.

Section 2 describes the background and related work. Section 3 gives the proposed algorithms. We compare the proposed algorithms with the literature in Section 4. Section 5 gives the conclusions.

2 Background and Related Work

2.1 Checkpointing and Snapshots

Fault-tolerance becomes very important for large-scale systems as the number of system components rises. For example, if the Mean Time Between Failures (MTBF) of a node is 10 years, then the MTBF of a 64,000 node system would be about 1.37 hours [2]. For fault-tolerance of long-lived computations (such as for number-crunching scientific problems), the age-old checkpoint-and-restart mechanism is very attractive [17]. The high failure rate puts additional pressure on the checkpoint mechanisms.

There are many mechanisms for taking checkpoints in distributed systems; two broad categories are coordinated and uncoordinated checkpointing. In uncoordinated checkpointing, processes autonomously take a checkpoint. The drawback of uncoordinated checkpoints is that checkpoints may be wasted because they are not consistent with checkpoints taken by other nodes. Further, multiple checkpoints need to be recorded at each process because the latest checkpoint at a process may not be consistent with the latest checkpoints at other process. Worse, this requires rollback to a much older system state and redoing computation from that older system state. In coordinated checkpointing, the nodes collectively record a consistent global state (i.e., snapshot) of the system so that no checkpoint is wasted. Coordinated checkpointing based on recording global snapshots is the preferred way of checkpointing in the high-performance massively parallel systems community [2], [10], [11], [14], [23], [34], [53], [61].

2.2 Related Work on Snapshots over Non-FIFO Channels

The Lai-Yang algorithm [39] works as follows:

1. Each process is initially white and turns red while taking a local snapshot.
2. A white (red) process sends white (red) colored messages.
3. Each process takes a local snapshot at any time before receiving a red message.

Each process keeps a log of all messages sent and received along each incident channel. After the local snapshots are collected at an initiator, the in-transit messages $SC_{i,j}$ are computed as the set-theoretic difference per channel $C_{i,j}$.

The Chandy-Lamport algorithm for a FIFO system and its variant by Mattern for a non-FIFO system [48] use a marker per logical channel. The role of a marker is threefold.

1. To inform processors that some processor has initiated the snapshot execution.
2. To distinguish white (prerecording) messages from red (postrecording) messages.
3. To mark the end of the white messages. In a system with non-FIFO channels, the computation messages are explicitly colored. To determine the number of white messages to be expected, Mattern's variant of the Chandy-Lamport algorithm works as follows [48]: It piggybacks the number of white messages sent along the channel on the corresponding marker sent on that channel. This allows the receiver to know how many white messages to expect before termination. The white messages are then reported by the receiver to the initiator. We name this algorithm as *piggyback*, in contrast to the *deficiency counting* and *vector counter* algorithms also introduced by Mattern [48].

The *deficiency counting* algorithm works as follows [48]: The algorithm uses the white/red coloring and does not use markers, analogous to [39]. Each processor keeps a counter that counts the number of messages sent minus the number of messages received. The counter gets recorded as part of the local state recording. When the initiator collects the local

snapshots, it knows how many white messages are in transit to the processors in the snapshot. Each red processor is required to report to the initiator each in-transit (white) message that is received. This continues until the deficit count is matched at the initiator.

The *vector counter* algorithm works as follows [48]: The algorithm uses the white/red coloring and does not use markers, analogous to [39]. This is a two-phase algorithm, where two waves are executed on any topology. For ease of exposition, a ring is assumed. Each processor P_i maintains a vector $V_i[1..n]$, where $V_i[j]$ tracks the number of white messages it sent to P_j. $V_i[i]$ gives the negative of the number of white messages received by P_i from all other processors. A control message with a control vector $C[1..n]$ circulates around the ring, performing $C \longleftarrow C + V_i; V_i \longleftarrow 0$ (for all components of the vectors). In this round, a white processor is also colored red, and the local snapshots are collected along with C_i. After the first round completes, $C[i]$ indicates the number of white messages that are in transit to P_i in the global snapshot. In the second round of the control message, the control message waits at each P_i until $V_i[i] + C[i] \leq 0$, i.e., all the white in-transit messages to P_i have been received. In the second round, relevant information about the in-transit messages is also collected.

The snapshot algorithms above can be repeatedly executed by introducing a third color and using the three colors cyclically [48]. (This technique can be adopted by our algorithms also to repeatedly execute the proposed snapshot algorithms.)

The two-dimensional grid-based algorithm by Garg et al. [23], [24] uses $O(n^{3/2})$ messages, each of size $\sqrt{n}$, by assuming a logical grid overlay on the underlying architecture and by using message coloring. Furthermore, the roles of the different processors are asymmetric. Each processor maintains a vector $white\_sent[1..n]$, where $white\_sent[j]$ gives the number of white messages sent to processor P_j. The grid-based algorithm performs accumulation of the $white\_sent$ vectors along the grid diagonal, and the diagonal elements then distribute the values to non-diagonal elements.

The centralized and tree-based algorithms by Garg et al. [23], [24] are based on deficit counting of the sum of in-transit messages, as used by Mattern in his *deficiency counting* algorithm [48]. The initiator computes the system-wide deficit, i.e., the total number of white messages in transit to all the processors in the snapshot. Let M be the total deficit. The initiator distributes M tokens throughout the system. The arrival of an in-transit message consumes a token. The tree-based algorithm is a round-based algorithm which guarantees that in each round, the global deficit is halved. Thus, there are at most $\lceil \log (M/n) \rceil$ rounds. The algorithm uses a tree structure, local coloring of processors to relate the number of in-transit messages received to the number of tokens available, and color transitions, to guarantee some invariants which give the bound on the number of rounds. Each round requires $O(n \log n)$ messages. The centralized is similar to the tree-based algorithm; it differs in the manner in which the control messages are exchanged and processed. Each of the $O(\log(M/n))$ rounds requires $O(n)$ messages. We use m to denote M/n.

2.3 Tree and Hypercube Overlays

In this paper, we propose the *simple_tree* and *hypercube* algorithms.

The *simple_tree* algorithm uses a tree overlay over the system. A tree overlay is easy to realize in clusters and large-scale supercomputers; as well as in large-scale dynamic peer-to-peer systems.

The *hypercube* algorithm uses a hypercube overlay over the system. A hypercube overlay can be easily implemented on clusters and supercomputers because they are under control by a management system that can instantiate a clear initial knowledge. The MPI library facilitates the use of interprocess communication over the hypercube overlay [62]. Figueira and Reddi showed how to build hypercube structures in heterogeneous networks [22].

A hypercube overlay can also be implemented over a P2P network. The survey by Risson and Moors [56] refers to some of these efforts; some other efforts are highlighted next. The pioneering P2P work by Plaxton et al. [54], Pastry [57], and Tapestry [67] all used hypercube routing. This hypercube routing has been studied to be effective under churn and failures [40], [42]. The Kad network that is accessed by the eMule client implemented the Kademlia protocol that relies on hypercubic networks [49]. Kuhn et al. used the dynamic hypercube graph topology to show robustness of their P2P system against worst-case joins and leaves [38]; the same dynamic hypercube graph was used in the eQuus distributed hash table by Locher et al. [43]. Streaming media systems over P2P overlays have been based on hypercubic topologies [44], [45] and cloud computing services with P2P online storage have also been based on hypercubes [26]. Load balancing of multimedia content streaming distribution over a hypercube overlay in structured P2P networks is shown by Han [30]. A very robust P2P system based on skip graphs [4], which are variants of hypercubes, is given in [32]. Naor and Weider presented a simple but general approach to build dynamic hypercubic P2P topologies [51]. Han showed how to build a hypercube overlay in a distributed setting [29], and Ren et al. showed how to provide a hypercube-based P2P information service for the data grid [55]. Hypercubes have been explicitly used in dynamic P2P networks by Schlosser et al. [59], [60]. Adler et al. showed that the hypercube-topology-based distributed hash table achieves asymptotically optimal load balance, among other nice logarithmic properties [1]. The hypercube topology for P2P systems was shown to be highly desirable for genealogical research [15]. The performance of approximate queries on a P2P network was studied on a hypercube overlay in [50]. Anceaume et al. designed a hypercube-based P2P overlay that is robust against Byzantine collusion and churn [3]. Hypercube-based P2P data stores were formally studied [19] and their churn resistance has also been studied [18], [20]. A tag-based system for managing XML data in a hypercubic overlay network was studied by Li et al. [41]. An open P2P architecture for XML message exchange was designed based on the hypercube network [58]. A leader election algorithm for P2P applications was presented for a hypercube overlay by Han and Xia [28]. A mechanism for wildcard search in structured P2P networks

int $white\_recd_i$;
int $white\_sent_i[1..n]$, $SENT_i[1..n]$;
state LS_i;
set of messages $SC_{j,i}$ for all j;

(1) When a white process P_i wants to initiate snapshot recording:
Broadcast a RECORD control message along a preestablished spanning tree and to P_i itself.

(2) When a white process P_i receives a RECORD control message or a red computation message:
turn red;
if a RECORD control message was received **then**
    propagate the RECORD control message along the spanning tree;
record local state LS_i;
$white\_recd_i$ is number of (white) messages received until now;
for $j \longleftarrow 1$ **to** n **do**
    $SENT_i[j] \longleftarrow white\_sent_i[j]$;
initialize $SC_{j,i}$ (for all $j \in N$) to $\emptyset$;
for $count \longleftarrow d-1$ **down to** 0 **do**
    send $SENT_i[j]$ to $P_{i \oplus 2^{count}}$, for all j such that
    $j = d - count - 1$ MSBs of $i \cdot \overline{(count+1)th \text{ LSB of } i} \cdot \underbrace{**\ldots*}_{count \text{ LSB bits}}$;
    receive 2^{count} entries of the form $SENT_*[k]$ from $P_{i \oplus 2^{count}}$;
    for all received entries of the form $SENT_*[k]$ **do**
        $SENT_i[k] \longleftarrow SENT_i[k] + SENT_*[k]$;
if $white\_recd_i = SENT_i[i]$ **then**
    local snapshot recording is complete.

(3) When a red process P_i receives a white message msg along $C_{j,i}$:
record msg in $SC_{j,i}$ as $SC_{j,i} \longleftarrow SC_{j,i} \cup \{msg\}$;
$white\_recd_i + +$;
if $white\_recd_i = SENT_i[i]$ and Step (2) is completed **then**
    local snapshot recording is complete.

Fig. 1. *Hypercube* algorithm for snapshot recording. Code is shown for processor P_i. $\oplus$ is the XOR operator.

assuming a logical hypercube overlay was proposed by Joung and Yang [33]. A symmetric VoIP conferencing network that allows for multigroup communication in P2P networks was presented for the hypercube overlay by Berndt [8].

3 Snapshot Algorithms

3.1 *Hypercube* Algorithm

We assume a hypercube overlay topology on the distributed system. Let $n = 2^d$. A hypercube overlay can be easily implemented; see Section 2.3. For convenience, we assume a preestablished spanning tree. A minimum spanning tree can be set up at a one-time cost of $O(n \log n)$ time and $O(|L| \log n)$ messages. We also assume that a single process runs at each processor as part of the distributed application.

Logically, a process can be in one of two states: white (prerecording) or red (postrecording). All processes are initially white. Application messages sent by a white process are colored white (prerecording messages). When some process records its local state, the algorithm is initiated. To inform other processes of this, a broadcast is done using RECORD control messages on a precomputed spanning tree. On receiving a RECORD message or a red computation message, a (white) process atomically records its local state (if it has not already done so) and turns red. Application messages sent by a red process are colored red (postrecording messages).

Three roles of a marker were described in Section 2.2. The *hypercube* algorithm does not use markers. However, the use of RECORD and red messages fulfills the first role of the marker. The coloring of messages fulfills the second role of the marker.

The third role of the marker is fulfilled by letting each process know the number of white messages sent to it. Rather than using a marker, this is achieved indirectly based on the following observation [48]: it is sufficient to know the total number of white messages sent to a process by all other processes. This number can be conveyed to a process using less than n messages, i.e., by not requiring a dedicated message from every other process. The proposed distributed algorithm can achieve this in $n \log n$ messages, wherein each process sends $\log n$ messages. (The length of the messages is analyzed in Section 3.1.2.) Specifically, we

use the hypercube overlay and perform n reductions concurrently in $\log\ n$ iterations.

There are three steps in the algorithm which is shown in Fig. 1.

1. Snapshot initiation: The snapshot initiator triggers a one-to-all broadcast of RECORD control messages. The RECORD messages can be sent along a preestablished spanning tree.
2. On receiving the RECORD message or a red colored message, the process records the local state and turns from white to red. It initializes the states of all the incoming channels to the empty set. (Henceforth, a red process sends red-labeled computation messages.) The algorithm then conveys the sum of the number of all white messages sent by all the processes to x, to that x, for every process x. The symmetrical manner in which this is achieved is the main innovation here.

   Each process P_i maintains $white\_sent_i[1..n]$ to count in $white\_sent_i[j]$ the number of white messages it sent to process P_j. $SENT_i[1..n]$ is initialized to $white\_sent_i[1..n]$. Using a hypercube overlay, the algorithm performs n all-to-one reductions concurrently in $\log\ n$ iterations. Each concurrent reduction is an in-network aggregation of the number of messages sent to a particular destination P_i. The in-network aggregation for P_i happens on a logical convergecast tree rooted at P_i and based on the order of the dimensions in the hypercube, from the MSB dimension to the LSB dimension. With respect to any destination P_i, the partial sum of the count of white messages sent to P_i exists in a hypercube that keeps halving in size in each of the $\log\ n$ iterations. In iteration $count$, where $count$ ranges from $d-1$ to 0, P_i communicates to $P_{i\oplus 2^{count}}$ the entries $SENT_i[j]$, for all j satisfying the following. Process j lies in the half-hypercube where j's label differs from i's label in the $(count+1)$th LSB and the $d-count-1$ MSBs match those of i's label. These j are characterized by

$$f(i, count) = d - count - 1 \text{ MSBs of } i \cdot \overline{(count+1)th \text{ LSB of } i} \cdot \underbrace{*\,*\,\ldots\,*}_{count \text{ LSB bits}}.$$

   At the end of $\log\ n$ iterations, the sum of the number of white messages sent to P_i is accumulated in $SENT_i[i]$, i.e., $\sum_{j\in N} white\_sent_j[i] = SENT_i[i]$. If $white\_recd_i = SENT_i[i]$, the algorithm terminates locally.

   Observe that the processes are implicitly synchronized [5] across the **for** loop of the variable $count$. Also, observe that a white process can receive a message of the form $SENT_*$. For simplicity and ease of exposition, this message is kept in the buffer and not processed while the process is white.
3. Recording channel states: When a white message is received from P_j by a red process P_i, it is added to the state of channel $C_{j,i}$ and the count $white\_recd_i$ is incremented. When step (2) is completed and $white\_recd_i$ equals $SENT_i[i]$, all white messages have been received, and the algorithm can terminate locally.

Optionally, if the snapshot needs to be assembled, a convergecast on a spanning tree can be performed after the termination of the local snapshot recording at each process. This also detects global termination of the algorithm. For checkpointing in large-scale systems, the checkpoints may be stored locally.

3.1.1 Correctness

The correctness of the local state recording is evident because we adapt Mattern's algorithm [48]. We only need to show that the channel states correctly record the in-transit white messages. For all $j \in N$, consider the n initial entries $white\_sent_j[i]$ and the logical convergecast tree rooted at P_i. The sum of these n initial entries represents the number of white messages sent to process P_i. In iteration $count$ ($0 \leq count \leq d-1$) of the main loop of step (2), 2^{count} entries of this form get added concurrently at various processes along the convergecast tree rooted at P_i. The total number of additions after all the rounds is

$$\sum_{0}^{d-1} 2^{count} = 2^d - 1 = n - 1,$$

yielding the desired sum of the n numbers. $\sum_j white\_sent_j[i]$ is thus correctly computed. The channel recording terminates when $\sum_j white\_sent_j[i] = white\_recd_i$.

3.1.2 Complexity Analysis

Theorem 1. *The distributed snapshot algorithm in Fig. 1 requires* $\log\ n$ *messages to be sent by each process and the sum of the message sizes sent by all the processes is* $O(n^2)$.

Proof. There are $\log\ n = d$ iterations in the main step (2). In each iteration, one message is sent and received by each process. Hence, a total of $\log\ n$ messages are sent and received by each process.

The messages sent in the various iterations have different sizes. In iteration one, $n/2$ integers are sent and received by a process. In iteration two, $n/4$ integers are sent and received by a process. In iteration k, $n/2^k$ integers are sent and received by a process. Specifically, the total number of integers sent and received by each process over all iterations is

$$\sum_{k=1}^{\log\ n} \frac{n}{2^k} = n \sum_{1}^{d} \frac{1}{2^k} = n - 1.$$

□

Observe that the message sizes form a geometric series. The sum of the lengths of all the messages in bits is $32(n^2 - n)$.

3.2 *Simple_tree* Algorithm

This is a three-phase algorithm that assumes a spanning tree. The processes are arranged in a logical tree. The three phases are as follows:

1. A tree broadcast by the root initiates the recording of local states, and turning red of white processes. A process may already be red if it has received a red

TABLE 1
Comparison of Noninhibitory Snapshot Algorithms for Non-FIFO Channels

Algorithm	Number of messages	Total message space	Local space	Symmetric	Response time	Parallel communication time
Lai-Yang [39]	$O(n)$	unbounded	unbounded	No	$O(n)$	unbounded
Deficiency counting [48]	$O(n(1+m))$	$O(n(1+m))$	$O(1)$	No	$O(n+nm)$	$(n+nm)t_s + t_w(n+nm)$
Vector counter on a ring [48]	$2n$	$O(n^2)$	$O(n)$	No	$2n$	$(2n)t_s + t_w(2n^2)$
Piggyback [48]	$O(n^2)$	$O(n^2)$	$O(n)$	Yes	$O(n)$	$(n)t_s + t_w(n)$
Grid-based [23, 24]	$O(n^{3/2})$	$O(n^2)$	$O(n)$	No	$O(\sqrt{n})$	$O((3\sqrt{n}+1)t_s + t_w(2n+2\sqrt{n}))$
Tree-based [23, 24]	$O(n \log n \log m)$	$O(n \log n \log m)$	$O(1)$	No	$O(n \log n \log m)$	$O((n \log n \log m)(t_s + t_w))$
Centralized [23, 24]	$O(n \log m)$	$O(n \log m)$	$O(1)$	No	$O(n \log m)$	$O((n \log m)t_s + t_w(n \log m))$
Simple_tree	$3(n-1)$	$O(n^2)$	$O(n)$	No	$3 \log n$	$(3 \log n)t_s + t_w((2n+1) \log n)$
Hypercube	$n \log n + n - 1$	$O(n^2)$	$O(n)$	Yes	$\log n$	$(\log n)t_s + t_w(n)$

n is the total number of processes. m is the average number of messages in transit to each process (on its incident channels) in the snapshot. t_s is the local start-up time and local reception time per message. t_w is the transmission time per word. Constants can vary depending on implementation.

message. A red process simply propagates the broadcast along the tree. When a process records its local state, it also records the value of the vector $white\_sent[1..n]$.

2. After the broadcast completes, a convergecast (initiated by the leaves) accumulates the vector $white\_sent_j[1..n]$, for all j, at the root. After the convergecast completes, $white\_sent[k]$ in the vector at the root contains the count of white messages sent to process P_k.
3. A tree broadcast initiated by the root distributes the accumulated values of $white\_sent$ to the processes. The kth component of the accumulated vector gives the sum of the number of white messages sent to P_k. P_k waits to receive this number of white messages before terminating the snapshot recording of channel states.

4 Discussion

Table 1 compares the proposed algorithms, denoted as the *simple_tree* and *hypercube* algorithms, with other noninhibitory algorithms for non-FIFO channels. We compare with the Lai-Yang algorithm [39], the *deficiency counting*, *vector counter*, and *piggyback* algorithms [48], and the two-dimensional grid-based, tree-based, and centralized algorithms by Garg et al. [23], [24].

All the algorithms are compared against the following metrics: number of messages, total message space, local storage, whether the roles of the processes are symmetrical, response time (or latency), and parallel communication time. We define the roles of the processes to be *symmetrical* if the processes execute identical code. In a symmetrical algorithm, there is perfectly uniform distribution of workload, no bandwidth and processing bottlenecks, and greater elegance. *Response time* is defined as the net parallel time of the control messages (to record the local states and enable detection of in-transit messages) in the parallel algorithm, counting the processing time for a message as one unit. A more refined version of the response time metric is the *parallel communication time* [25]. Here, the time for a message is $t_s + t_w x$, where t_s is the local processing overhead per message (at the sender and the receiver), t_w is the transmission time per word, and x is the number of words in the message. The *parallel communication time* is the net parallel message time in the parallel algorithm.

4.1 Symmetry

The tree-based, grid-based, and the centralized algorithms [23], [24] all have varying degrees of asymmetry among the processes. Specifically, the grid-based algorithm performs accumulation of the $white\_sent$ vectors along the grid diagonal, and the diagonal elements then distribute the values to nondiagonal elements. The tree algorithms (*simple_tree* and tree-based) have asymmetrical roles among leaf nodes, internal nodes, and the root node. Note that in *vector counter*, the initiator plays the additional role of changing the phases of the algorithm; hence, we classify it as being asymmetric. The only distributed algorithms that have perfectly symmetric roles for the processes are the *piggyback* and *hypercube* algorithms. Observe from Table 1 that the proposed *hypercube* algorithm has the lowest number of messages from among algorithms in which the roles of all the processes are completely symmetrical.

4.2 Number of Messages

The *vector counter* algorithm and *simple_tree* use fewer messages than the *hypercube* algorithm. Notwithstanding the asymmetry of roles in the grid-based and tree-based algorithms [23], [24], the *hypercube* algorithm is also superior to the grid-based and tree-based algorithms in terms of the number of messages and in terms of response time/parallel communication time. As the system scales up in terms of the number of processors, the number of messages becomes very important. It is more efficient to send few large messages than more small messages.

4.3 Response Time

The response time of the *hypercube* algorithm is $\log\ n$ because the messages in step (2) are immediately pipelined after the RECORD messages. Hence, there is no latency for the initiation phase. Compared to the *simple_tree* algorithm, the *hypercube* algorithm has lower response time: $\log\ n$, as

against $3 \log n$ for the sequential convergecast and broadcast that follow the initiation phase in *simple_tree*. *Simple_tree* is asymmetric as it requires different roles to be played by leaf nodes, internal nodes, and the root node. The response time of the *hypercube* and *simple_tree* algorithms is $O(log\ n)$, which is better than that of all other algorithms.

4.4 Parallel Communication Time

This metric is a refined version of the response time metric. The parallel communication time of the hypercube is $(\log n)t_s + t_w(n)$. This is the lowest among all the algorithms, indicating that this is the fastest algorithm in terms of completing the snapshot recording globally. In a fast algorithm, the time window for the transmission of algorithm messages is lowered. Hence, the snapshot gets recorded earlier, subject to the arrival of the in-transit (white) messages.

The parallel communication time of the hypercube is lowest among all algorithms. We make the following conjectures, based on the properties of the hypercube architecture [25]:

1. Among distributed snapshot recording algorithms that are perfectly symmetrical, i.e., identical code is executed by the processes, the *hypercube* algorithm in Fig. 1 is optimal in terms of the number of messages used.
2. Among distributed snapshot recording algorithms, the *hypercube* algorithm in Fig. 1 is optimal in terms of parallel communication time.

Note, however, that the hypercube overlay may contain multiple edges compared to the tree overlay; this may impact the parallel communication time, as well as the *hypercube* property of a perfectly balanced workload and absence of any bottlenecks.

Ultimately, the performance of *hypercube* depends on how well the overlay is constructed. Despite this, we note that even in the case of a poorly constructed overlay, the parallel communication time is largely unaffected because of the cut-through routing that can be used in the overlay. The full expression for the parallel communication time is: $t_s + lt_h + t_w m$, per message. Here, t_h is the per hop communication time; l is the number of hops taken by the message and represents the dilation of the overlay embedding. As pointed out by Grama et al. [25], "t_h is dominated by the start-up latency t_s for small messages and by $t_w m$ for large messages. Since the maximum number of hops l in most networks is relatively small, the per-hop time can be ignored with little loss of accuracy."

In the hypercube overlay, when $n \neq 2^d$, some dummy nodes need to be added, which disturb the symmetry of roles in the perfect case when $n = 2^d$. However, many large MIMD machines and distributed systems have a number of nodes which is a round and even number if not a power of two.

There are two potential drawbacks of *simple_tree* and *hypercube*. Compared to the centralized algorithm [23], [24] and the tree-based algorithm (which is essentially a centralized $(\log m)$-phase algorithm) [23], [24], *simple_tree* and *hypercube* require $O(n)$ local storage instead of $O(1)$ local storage. This may not be a serious drawback because $O(n)$ data structures, such as for vector clocks [47], often need to be maintained. The total message space for *simple_tree* and *hypercube* may also be higher than for the essentially centralized algorithms [23], [24]. Despite these drawbacks, the algorithms offer the advantages of being fast and message-efficient. In addition, the *hypercube* algorithm is symmetrical and has greater potential for balanced workload and congestion-freedom than the algorithms in [23], [24].

5 Conclusions

A message overhead of $O(n^2)$ messages per snapshot becomes too expensive and is not scalable as the number of processors increases. Recent work has focused on reducing the snapshot complexity in such systems. Garg et al. proposed a grid-based, tree-based, and centralized algorithm to reduce this overhead [23], [24]. This paper presented two simple snapshot algorithms: *simple_tree* and *hypercube*, that 1) use fewer messages than the algorithms in [23], [24] and 2) have lower response time and parallel communication times than the algorithms in [23], [24]. In addition, the *hypercube* algorithm is symmetrical and has greater potential for balanced workload and congestion-freedom than the algorithms in [23], [24]. The algorithms find direct application in large-scale distributed systems such as peer-to-peer systems and MIMD supercomputers which have a fully connected topology of a large number of processors.

Acknowledgments

A preliminary version of this result appears as a poster paper [37].

References

[1] M. Adler, E. Halperin, R.M. Karp, and V.V. Vazirani, "A Stochastic Process on the Hypercube with Applications to Peer-to-Peer Networks," *Proc. ACM Symp. Theory of Computing (STOC),* pp. 575-584, 2003.

[2] S. Agarwal, R. Garg, M. Gupta, and J. Moreira, "Adaptive Incremental Checkpointing for Massively Parallel Systems," *Proc. Int'l Conf. Supercomputing,* pp. 277-286, 2004.

[3] E. Anceaume, R. Ludinard, A. Ravoaja, and F. Brasileiro, "PeerCube: A Hypercube-Based P2P Overlay Robust against Collusion and Churn," *Proc. Second Int'l Conf. Self-Adaptive and Self-Organizing Systems,* pp. 15-24, 2008.

[4] J. Aspnes and G. Shah, "Skip Graphs," *Proc. 14th Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 384-393, 2003.

[5] B. Awerbuch, "Complexity of Network Synchronization," *J. ACM,* vol. 32, no. 4, pp. 804-823, 1985.

[6] R. Baldoni, R. Jimenez-Peris, M. Patino-Martinez, L. Querzoni, and A. Virgillito, "Dynamic Quorums for DHT-Based Enterprise Infrastructures," *J. Parallel and Distributed Computing,* vol. 68, no. 9, pp. 1235-1249, 2008.

[7] R. Baldoni, F. Quaglia, and P. Fornara, "An Index-Based Checkpointing Algorithm for Autonomous Distributed Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 2, pp. 181-192, Feb. 1999.

[8] P. Berndt, "Using Symmetric Distributed Processing for Peer-to-Peer VoIP Conferencing in Auditory Virtual Environments," *Proc. Seventh Int'l Workshop Peer-to-Peer Systems (IPTPS),* 2008.

[9] W.J. Bolosky, J.R. Douceur, D. Ely, and M. Theimer, "Feasability of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs," *Proc. ACM SIGMETRICS '00,* pp. 34-43, 2000.

[10] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated Application-Level Checkpointing of MPI Programs," *Proc. Symp. Principles and Practice of Parallel Programming (PPoPP '03),* pp. 84-94, 2003.

[11] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective Operations in Application-Level Fault-Tolerant MPI," *Proc. Int'l Conf. Supercomputing,* pp. 234-243, 2003.

[12] G. Cao and M. Singhal, "On Coordinated Checkpointing in Distributed Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 12, pp. 1213-1225, Dec. 1998.

[13] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 1, pp. 63-75, 1985.

[14] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI," *Proc. Int'l Conf. Supercomputing '06,* Nov. 2006.

[15] R. Crowther and S. Woodfield, "Hypercubes: A Superior Topology for Real-Time Genealogical Collaboration Networks," *Proc. Family History Technology Workshop,* 2001.

[16] G. De Candia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles,* pp. 205-220, 2007.

[17] E. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys,* vol. 34, no. 3, pp. 375-408, Sept. 2002.

[18] D. Fahrenholtz and V. Turau, "Improving Churn Resistance of P2P Data Stores Based on the Hypercube," *Proc. Fifth Int'l Symp. Parallel and Distributed Computing,* pp. 263-270, 2006.

[19] D. Fahrenholtz and A. Wombacher, "A Formal Communication Model for Lookup Operations in a Hypercube-Based P2P Data Store," *Proc. First Int'l Conf. Collaborative Computing: Networking, Applications and Worksharing,* Dec. 2005.

[20] D. Fahrenholtz, "A Hypercube-Based Peer-to-Peer Data Store Resilient against Peer Population Fluctuation," PhD thesis, Hamburg Univ. of Technology, 2008.

[21] L. Fan, H. Taylor, and P. Trinder, "Design Issues for Peer-to-Peer Massively Multiplayer Online Games," *Proc. Second Int'l Workshop Massively Multiplayer Virtual Environments,* 2008.

[22] S. Figueira and V. Reddi, "Topology-Based Hypercube Structures for Global Communication in Heterogeneous Networks," *Proc. European Conf. Parallel Computing (Euro-Par '05),* pp. 994-1004, 2005.

[23] R. Garg, V. Garg, and Y. Sabharwal, "Scalable Algorithms for Global Snapshots in Distributed Systems," *Proc. 20th Ann. Conf. Supercomputing,* pp. 269-277, Nov. 2006.

[24] R. Garg, V. Garg, and Y. Sabharwal, "Efficient Algorithms for Global Snapshots in Large Distributed Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 21, no. 5, pp. 620-630, May 2010.

[25] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing,* second ed. Addison-Wesley, 2003.

[26] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer, "Havelaar: A Robust and Efficient Reputation System for Active Peer-to-Peer Systems," *Proc. First Workshop Economics of Networked Systems (NetEcon),* June 2006.

[27] T. Hampel, T. Bopp, and R. Hinn, "A Peer-to-Peer Architecture for Massive Multiplayer Online Games," *Proc. Fifth ACM SIGCOMM Workshop Network and System Support for Games,* 2006.

[28] S.C. Han and Y. Xia, "Optimal Leader Election Scheme for Peer-to-Peer Applications," *Proc. Sixth Int'l Conf. Networking,* 2007.

[29] S.C. Han, "Distributed Many-to-Many Mapping Algorithm in the Hypercube Network," *Proc. Fourth Int'l Conf. Networked Computing and Advanced Information Management,* 2008.

[30] S.C. Han, "Load-Balancing Content Distribution in Structured Peer-to-Peer Networks," *Proc. Fourth Int'l Conf. Networked Computing and Advanced Information Management,* 2008.

[31] J.-M. Helary, A. Mostefaoui, and M. Raynal, "Communication-Induced Determination of Consistent Snapshots," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 9, pp. 865-877, Sept. 1999.

[32] R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Taubig, "A Polylogarithmic Time Algorithm for Distributed Self-Stabilizing Skip Graphs," *Proc. 28th ACM Symp. Principles of Distributed Computing (PODC),* pp. 131-140, Aug. 2009.

[33] Y.-J. Joung and L.-W. Yang, "Wildcard Search in Structured Peer-to-Peer Networks," *IEEE Trans. Knowledge and Data Eng.,* vol. 19, no. 11, pp. 1524-1540, Nov. 2007.

[34] A. Kangarlou, P. Ruth, D. Xu, and P. Eugster, "Taking Snapshots of Virtual Networked Environments," *Proc. Virtualization Technologies in Distributed Computing Workshop '07,* Nov. 2007.

[35] A. Kshemkalyani, M. Raynal, and M. Singhal, "An Introduction to Snapshot Algorithms in Distributed Computing," *Distributed Systems Eng.,* vol. 2, no. 4, pp. 224-233, 1995.

[36] A. Kshemkalyani and B. Wu, "Detecting Arbitrary Stable Properties Using Efficient Snapshots," *IEEE Trans. Software Eng.,* vol. 33, no. 5, pp. 330-346, May 2007.

[37] A. Kshemkalyani, "A Symmetric *O(n log n)* Message Distributed Algorithm for Large-Scale Systems," *Proc. IEEE Int'l Cluster Computing Conf.,* 2009.

[38] F. Kuhn, S. Schmid, and R. Wattenhofer, "A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn," *Proc. Fourth Int'l Workshop Peer-To-Peer Systems (IPTPS),* pp. 13-23, Feb. 2005.

[39] T.-H. Lai and T. Yang, "On Distributed Snapshots," *Information Processing Letters,* vol. 25, no. 3, pp. 153-158, 1987.

[40] S.S. Lam and H. Liu, "Failure Recovery for Structured P2P Networks: Protocol Design and Performance Evaluation," *Computer Networks,* vol. 50, pp. 3083-3104, 2006.

[41] Y. Li, M.T. Ozsu, and K.-L. Tan, "XCube: Processing XPath Queries in a Hypercube Overlay Network," *Peer-to-Peer Networking Applications,* vol. 2, pp. 128-145, 2009.

[42] H. Liu and S.S. Lam, "Neighbour Table Construction and Update in a Dynamic Peer-to-Peer Network," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* 2003.

[43] T. Locher, S. Schmid, and R. Wattenhofer, "eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System," *Proc. Sixth IEEE Int'l Conf. Peer-to-Peer Computing (P2P),* pp. 3-11, Sept. 2006.

[44] T. Locher, R. Meier, S. Schmid, and R. Wattenhofer, "Push-to-Pull Peer-to-Peer Live Streaming," *Proc. 21st Int'l Symp. Distributed Computing (DISC),* pp. 388-402, Sept. 2007.

[45] T. Locher, R. Meier, S. Schmid, and R. Wattenhofer, "Robust Live Media Streaming in Swarms," *Proc. 19th Int'l Workshop Network and Operating Systems Support for Digital Audio and Video (NOSSDAV),* pp. 121-126, June 2009.

[46] D. Manivannan, R.H.B. Netzer, and M. Singhal, "Finding Consistent Global Checkpoints in a Distributed Computation," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 6, pp. 623-627, June 1997.

[47] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Workshop Parallel and Distributed Algorithms,* M. Cosnard, ed., pp. 215-226, 1988.

[48] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *J. Parallel and Distributed Computing,* vol. 18, no. 4, pp. 423-434, 1993.

[49] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," *Proc. First Int'l Workshop Peer-to-Peer Systems (IPTPS),* pp. 53-65, 2002.

[50] A. Mowat, R. Schmidt, M. Schumacher, and I. Constantinescu, "Extending Peer-to-Peer Networks for Approximate Search," *Proc. 23rd ACM Symp. Applied Computing,* pp. 455-459, 2008.

[51] M. Naor and U. Wieder, "Novel Achitectures for P2P Applications: The Continuous-Discrete Approach," *ACM Trans. Algorithms,* vol. 3, no. 3, 2007.

[52] L. Ni, A. Harwood, and P.J. Stuckey, "Realizing the E-Science Desktop Peer Using a Peer-to-Peer Distributed Virtual Machine Middleware," *Proc. Fourth Int'l Workshop Middleware for Grid Computing (MCG '06),* pp. 7-12, 2006.

[53] A. Oliner, L. Rudolph, and R. Sahoo, "Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability," *Proc. Int'l Conf. Supercomputing '06,* pp. 14-23, 2006.

[54] G. Plaxton, R. Rajaraman, and A. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," *Proc. ACM Symp. Parallel Architectures and Algorithms,* pp. 311-320, June 1997.

[55] H. Ren, Z. Wang, and Z. Liu, "A Hypercube Based P2P Information Service for Data Grid," *Proc. Fifth Int'l Conf. Grid and Cooperative Computing,* pp. 508-513, 2006.

[56] J. Risson and T. Moors, "Survey of Research towards Robust Peer-to-Peer Networks: Search Methods," *Computer Networks,* vol. 50, no. 17, pp. 3485-3521, 2006.

[57] A. Rowstron and P. Druschel, "Pastry: Scalable Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms,* pp. 329-350, Nov. 2001.

[58] B. Schandl, "OPAX—An Open Peer-to-Peer Architecture for XML Message Exchange," Diploma thesis, Univ. of Vienna, 2004.

[59] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl, "Ontology-Based Search and Broadcast in HyperCuP," *Proc. Int'l Semantic Web Conf.,* 2002.

[60] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl, "HyperCuP—Hypercubes, Ontologies, and Efficient Search on Peer-to-Peer Networks," *Agents and Peer-to-Peer Computing,* pp. 112-124, Springer, 2003.

[61] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Implementation and Evaluation of a Scalable Application-Level Checkpoint-Recovery Scheme for MPI Programs," *Proc. Int'l Conf. Supercomputing '04,* Nov. 2004.

[62] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference.* MIT Press, 1996.
[63] C. Tang, R.N. Chang, and E. So, "A Distributed Service Management Infrastructure for Enterprise Data Centers Based on Peer-to-Peer Technology," *Proc. IEEE Int'l Conf. Services Computing,* pp. 52-59, 2006.
[64] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A Scalable Application Placement Controller for Enterprise Data Centers," *Proc. 16th Int'l Conf. World Wide Web (WWW),* C.L. Williamson, M.E. Zurko, P.F. Patel-Schneider, and P.J. Shenoy, eds., pp. 331-340, 2007.
[65] S. Venkatesan, "Message-Optimal Incremental Snapshots," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* pp. 53-60, 1989.
[66] Top 500 Supercomputer Sites, <http://www.top500.org>, 2009.
[67] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz, "Tapestry: A Resilient Global-Scale Overlay for Service Deployment," *IEEE J. Selected Areas in Comm.,* vol. 22, no. 1, pp. 41-53, Jan. 2004.

Ajay D. Kshemkalyani received the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987, and the PhD degree in computer and information science from The Ohio State University in 1991. He is a professor of computer science at the University of Illinois at Chicago. Previously, he spent several years at IBM Research Triangle Park working on various aspects of computer networks. His research interests are in computer networks, distributed computing, algorithms, and concurrent systems. In 1999, he received the National Science Foundation's CAREER Award. He is currently on the Editorial Board of the Elsevier Journal, *Computer Networks*. He is a distinguished member of the ACM and a senior member of the IEEE.