The Essence of Distributed Snapshots

K. Mani Chandy

Computer Science Department
California Institute of Technology

Caltech-CS-TR-89-5

The Essence of Distributed Snapshots

K. Mani Chandy*
California Institute of Technology

6 March 1989

1 Introduction

A distributed system has no global clock, and it is the absence of a global clock that makes for several interesting problems, one of which is obviously important, but apparently trivial: 'Record the state of the system.' Recording the state of distributed system is called 'taking a global snapshot' after [2]. If there were a clock, taking global snapshots would be straightforward: Each process records its state or at some predetermined time, and the collection of recorded process states is used to construct a system state.

Global snapshots are useful in a variety of situations [2,3,6]. The goal of this paper is to identify the essential properties of global snapshots so as to simplify proofs of global snapshot algorithms and to aid in the design of new algorithms.

2 A Distributed System

2.1 Standard Definitions

We shall first define a distributed system as in [8].

*On leave of absence from the University of Texas at Austin.

A prefix of a sequence z is an initial subsequence of z. A prefix-closed set of sequences is a set such that every prefix of a sequence in the set is also in the set.

A system is a set of *components*. A component is a set of *events* and a prefix-closed set of sequences of its events called its set of *computations*.

A *projection* of a sequence v on a component is the sequence obtained from v by deleting all events in v that are not events of the component.

A *system computation* is a sequence v of events of components of the system such that the projection of v on each component of the system is a computation of that component.

Let $w.p$ be a computation of component p, all p. Let P be a set of components. An *interleaving* of a set of component computations $\{w.p \,|\, p \in P\}$ is a sequence, v, of events of components in P, such that the projection of v on p is $w.p$, all $p \in P$.

We use (y, z) for the catenation of sequences y and z.

2.2 Processes and Channels

A component of a distributed system is either a *process* or a *channel*. Distinct processes have disjoint sets of events, and distinct channels have disjoint sets of events.

A channel is *used by* exactly two processes. The events of a channel are events of the processes that use the channel. We shall restrict attention to channels that satisfy the following monotonicity condition.

Let c be a channel used by processes q and r. Let u, v be computations of c, where $u.r = v.r$, and $u.q$ is a prefix of $v.q$. Let e be an event on r.

A Monotonicity Property If (u, e) is a computation of c, then (v, e) is also a computation of c.

Explanantion The monotonicity condition implies that the execution of events on one process cannot inhibit the execution of an event on another process. If a channel c is used by processes q and r, and there is a computation of c in which e is executed on process r after q and r have executed computations a and b respectively, then there is a computation

of c in which e is executed after r has executed b, and q has executed *an arbitrary sequence of events following a.*

Example: Bounded First-In-First-Out Buffers Consider a first-in-first-out buffer, with a capacity of N messages ($N > 0$), shared by a single producer process and a single consumer process. Such a buffer is a channel that has the monotonicity property, as shown by the following informal argument.

The producer can append any message to the buffer if the buffer is not full. The consumer can receive a message m from the buffer if the buffer is not empty and m is the message at the head of the buffer queue. If the producer can produce a message after it has produced i messages and the consumer has consumed j messages, then the producer can produce a message after it has produced i messages, and the consumer has consumed more than j messages. Therefore, the monotonicity property holds with r as the producer and q as the consumer.

By a similar argument, additional production does not prevent the consumer from receiving the message at the head of the buffer; the monotonicity property also holds with r as the consumer and q as the producer. Therefore, the channel has the monotonicity property.

Example: Stacks Next consider a channel which is a stack. Let m be the message at the top of the stack, if the stack is not empty. The consumer can execute the event: Pop the stack and consume m. The producer can execute an event: Push a message m' on top of the stack. Such a buffer does not have the monotonicity property because an event of the producer — push m' on the stack — where $m \neq m'$, can prevent the consumer from executing the event: Pop the stack and receive message m. Therefore, an event on one process can inhibit the execution of an event on the other.

Note: Symmetry of Processes One way of defining channels is in terms of causality: one of the processes sends a message on the channel, and the other receives the message, thus there is a causal relationship between the sending and the receiving of the message. The definition of channels used in this paper is symmetric with respect to both processes; the defini-

tion does not employ the concept of causality. Monotonicity appears to be an important property of channels of distributed systems.

3 The Problem

Restrict attention to one given system. Let z be a finite computation of the system. For ease of exposition, assume that all events in z are distinct. (If events are repeated in z, then number the events, so that the pair — event-name, number — is distinct.) Let $z.p$ be the projection of z on a process p. Let $x.p$ be any prefix of $z.p$. Let S be the set of process computations $\{x.p | p \text{ is a process } \}$.

Set, S, is defined to be a global snapshot in z if and only if there exists a system computation y where:

1. y is an interleaving of the set of process computations $z.p$, and
2. every event in S occurs in y before every event that is not in S.

The problem is to determine simple necessary and sufficient conditions for S to be a global snapshot in z.

Motivation for the Problem Set S is a global snapshot in z if and only if there is a system computation that first takes the system to a state in which each process p has executed $x.p$, and then to the state in which each process p has executed $z.p$. Informally, S is a global snapshot in z if and only if it *could have* happened that all events in S were executed before all events that are not in S. If S is a global snapshot in z, then properties about S can be used to deduce properties about the state of the system after z is executed. Therefore, it is helpful to determine simple conditions that guarantee that S is a global snapshot.

4 A Solution

The obvious algorithm to determine if S is a global snapshot in z is as follows: Since z is finite, enumerate all interleavings of $z.p$, and determine if there is one with the desired properties. This approach is computationally

intractable if the number of processes is large. Next, we present a theorem that helps us to design tractable solutions.

4.1 Compatible Computations

Let c be a channel. Let c be used by processes q and r. Let u and v be computations of q and r respectively. Process computations u and v are defined to be *compatible with respect to* c if and only if there exists an interleaving w of u and v such that the projection of w on c is a computation of c.

Informally, u and v are compatible with respect to c if and only if process computations u and v could have occurred in a computation of a system consisting of only the two processes q and r, and the single channel c.

Example: Bounded First-In-First-Out Buffers Let c be a channel that is a first-in-first-out buffer with a capacity of N where $N > 0$, and where the buffer is initially empty. Let u and v be computations of the processes that send and receive (respectively) on the channel. Then u and v are compatible with respect to c if and only if the sequence of messages received along c in v is a prefix of the sequence of messages sent along c in u, and the number of messages sent along c in u exceeds the number of messages received along c in v by at most N.

Let z and $x.p$ be as in the problem definition, i.e., z is a system computation and $x.p$ is a prefix of $z.p$. Let the producer and consumer for c be q and r respectively. Since z is a system computation, the sequence of receives along c in $z.r$ is a prefix of the sequence of sends along c in $z.q$. Therefore, the sequence of receives along c in $x.r$ is a prefix of the sequence of sends along c in $x.q$ if and only if the number of receives along c in $x.r$ is at most the number of sends along c in $x.q$. Therefore $x.q$ and $x.r$ are compatible with respect to c if and only if

$$0 \leq (nsends - nreceives) \leq N$$

where *nsends* and *nreceives* are the numbers of sends and receives along channel c in $x.q$ and $x.r$ respectively.

4.2 The Snapshot Theorem and its Applications

We shall first give the theorem, discuss its consequences, and then prove it. Let z, $z.p$, $x.p$ and S be as given earlier.

Theorem Set, S, is a global snapshot in z if and only if, for each channel, c:
$x.q$ and $x.r$ are compatible with respect to c, where q and r are the processes that use r.

Applications of the Theorem The proof that S is a global snapshot of an arbitrary system reduces to a proof of compatibility of a pair of process computations for each channel. Let us use this fact in developing algorithms for a couple of problems. The following discussion is very brief and informal, because our goal is only to demonstrate the use of the theorem, rather than to give a thorough exposition of the algorithms.

The Snapshot Algorithm We shall develop the algorithm in [2]. Consider a system in which channels are first-in-first-out and the capacity of a channel is arbitrarily large. Initially all channels are empty. We wish to develop a distributed algorithm to record the global state of the system.

Consider a channel c used by processes q and r, where q sends along c, and r receives along c, and initially c is empty. As discussed earlier, process computations $x.q$ and $x.r$ are compatible with respect to c if and only if the number of receives along c in $x.r$ is at most the number of sends along c in $x.q$. Therefore the problem of algorithm design reduces to this: Guarantee that the number of receives before the receiver records its state is at most the number of sends before the sender records its state, and also guarantee that every process records its state eventually.

One way of doing this is as follows: After a process records its state, it sends a special message called a *marker* along each of its outgoing edges. On receiving a marker a process records its state if it has not done so. At least one process (called the initiator) records its state in finite time; if there is a path (a sequence of channels) from the initiator to all other processes then every process records its state in finite time of the intiator.

Logical Time Consider the same system as in the previous paragraph. Let z be a computation of the system. We are required to give each event in z a unique number, called its logical time, such that the set of events with logical times less than n corresponds to a global snapshot in z, for all n. Let $x.p$ be the prefix of $z.p$ consisting of all events with logical times less than n; we require that the set S (defined as before as $\{x.p\}$) be a global snapshot.

As in the previous problem, the problem of algorithm design reduces to this: Guarantee that for each channel c, the number of messages received along c in $x.r$ is at most the number of messages sent along c in $x.q$, where q and r are the processes that send and receive along c, respectively. This is equivalent to: guarantee that logical times of events are such that the event of receiving a message has a higher logical time than the event of sending that message. One way of doing so is in [7]: put a time-stamp t on each message where t is the logical time of the event that sends the message, and give the event that receives a message a logical time that is greater than the time-stamp of the message.

(The goal for logical time in the seminal paper [7] is different from that given here, because it is based on the concept of causality. Our goal here is limited: to demonstrate a use of the snapshot theorem.)

4.3 Proof of The Snapshot Theorem

Snapshot Theorem Let z be a finite computation of the system. Let $x.p$ be a prefix of $z.p$, all p. Let S be the set of process computations $\{x.p | p$ is a process $\}$. Set S is a global snapshot in z if and only if, for each channel c, computations $x.q$ and $x.r$ are compatible with respect to c, where q and r are the processes that use c.

Proof If $x.q$ and $x.r$ are incompatible with respect to c, then there is no interleaving of $x.q$ and $x.r$ that is a computation of c, and hence S is not a global snapshot. Next, we prove that if for each c, $x.q$ and $x.r$ are compatible with respect to c, where q, r use c, then S is a global snapshot. The proof given here is a generalization of the proofs given in [2,5] which are limited to unbounded first-in-first-out channels.

Define sequence y as follows: y is the permutation of z where all events in S occur before all events that are not in S, and apart from this change, the order of events in z is retained in y. We shall prove that S is a global snapshot by proving that y is a system computation.

Let w be a permutation of z. We shall give an algorithm which starts with $w = z$ and that ends with $w = y$, and where the algorithm maintains the invariant: w is a system computation.

The Algorithm Initially $w = z$. While w contains a pair of adjacent elements d and e, where d occurs before e, and d is not in S, and e is in S: interchange the positions of d and e in w.

Proof of Termination We prove that the algorithm terminates in a finite number of steps by using the metric: the number of pairs (f, g), where event f occurs earlier than event g in w, and f is not in S, and g is in S. The algorithm terminates if and only if the metric is zero, in which case $w = y$.

The metric has a finite value initially, and every step decreases it; hence the algorithm terminates in a finite number of steps.

Proof of the Invariant We prove the stronger invariant that $w.p = z.p$, all processes p, and $w.c$ is a computation of c, all channels c, where $w.d$ and $z.d$ are the projections of w and z, respectively, on component d. The invariant holds initially, because $w = z$. Let w' be the same as w except that d and e are interchanged. Our proof obligation is to show that w' satisfies the invariant if w satisfies it.

Since $x.q$ is a prefix of $z.q$ and since $w.q = z.q$, it follows that $x.q$ is a prefix of $w.q$. Therefore, if two adjacent events in w are on the same process, q, and the first of the two events is not in $x.q$, then the second is not in $x.q$ either. Since d is not in S and e is in S, it follows that d and e cannot be on the same process. Let d be on process q and let e be on process r, where $r \neq q$.

Since d and e are on different processes, $w'.p = w.p$, all p, and therefore $w'.p = z.p$.

If d and e are on different channels, then the projections of w and w' on each channel are identical, and hence the invariant holds for w'. Therefore, we need only consider the case where d and e are on the same channel; let this channel be c. Our only remaining proof obligation is to show that $w'.c$ is a computation of c.

Let t be the prefix of w preceding d in w. Then (t, d, e) is a prefix of w, and (t, e, d) is a prefix of w'.

Since $x.q$ and $x.r$ are compatible with respect to c, there exists an interleaving h of $x.q$ and $x.r$ such that the projection of h on c is a computation of c. Event e is in $x.r$, and therefore is in h. Define u as the prefix of h preceding e. Therefore, (u, e) is a prefix of h, and hence it is a computation of c. Since both $u.r$ and $t.r$ are the prefixes of $x.r$ that precede e, it follows that $u.r = t.r$. Since d is not in $x.q$, it follows that $x.q$ is a prefix of $t.q$. Since $u.q$ is a prefix of $x.q$, it follows from the transitivity of the prefix relation that $u.q$ is a prefix of $t.q$. From the monotonicity property, the projection of (t, e) on c is a computation of c.

Applying the monotonicity property again, the projection of (t, e, d) on c is a computation of c, since the projections of (t, d) and (t, e) on c are computations of c.

Let m be the length of the sequence (t, e, d). We shall prove by induction on k, that for $k \geq m$: $g'.c$ is a computation of c where g' is the prefix of w' of length k.

Base Case: $k = m$. This case has already been proved.

Induction Step: Let f be the $(k+1)$-th event in w. Our proof obligation is to show that the projection of (g', f) on c is a computation of c. Let g be the prefix of w of length k. The projection of (g, f) on c is a computation of c because (g, f) is a prefix of w. From the induction hypothesis, the projection of g' on c is a computation of c. For $k \geq m$: $g.q = g'.q$ and $g.r = g'.r$. If f is on c, the from the monotonicity property of c, the projection of (g', f) on c is a computation of c. If f is not on c, then the projection of (g', f) on c is the same as the projection of g' on c, and the result follows.

5 Partial Snapshots

There are some problems in which a snapshot of some subset of processes and channels is useful, and a global snapshot of all processes and channels is not necessary. We define a *partial* snapshot of a set of processes, Q, in a manner analogous to the definition of a global snapshot. Let z be a system computation. Let $x.p$ be a prefix of $z.p$. Let S be the set of process computations $\{x.q | q \in Q\}$. Set S is defined to be a partial snapshot in z if and only if there exists a system computation y where:

1. y is an interleaving of the set of process computations $z.p$, all processes p, and

2. for each process q in Q, the events in $x.q$ appear in y before the events of q that are not in $x.q$.

A partial snapshot is a global snapshot if Q is the set of all processes.

Next, we shall define a class of problems for which partial snapshots are helpful.

5.1 Termination Problems

Let w be a system computation. Set, Q, is defined to have terminated after w if and only if,

1. for all events e, and all processes q in Q, if $(w.q, e)$ is a computation of q, then e is an event on a channel between q and a process in Q, and

2. for all channels c between processes in Q, there is no event e such that $(w.c, e)$ is a computation of c.

Informally, the first condition says that after a process q has executed $w.q$ it can only execute events on channels connecting it to other processes in Q. The second condition says that there is no extension of a computation of a channel c between processes in Q after w. The two conditions, together, imply that the processes in Q cannot execute events after system computation w.

Example: Full-Buffer Deadlock Consider a system in which each channel is a buffer with a capacity of N, where $N > 0$. A process is either waiting or active. A waiting process is waiting to send a message on any one of a set of full outgoing channels (i.e., channels containing N messages); a waiting process continues to wait until at least one of the channels that it is waiting for becomes not full, and it then sends a message on that channel and becomes active. Waiting processes do not receive messages. A set of processes, Q, is said to be deadlocked if and only if:

1. each channel between processes in Q is full (or equivalently, the number of messages sent on the channel exceeds the number of messages received on the channel by N), and

2. each process in Q is waiting to send messages only along channels to other processes in Q.

The problem is to detect a deadlocked state.

A dual of this problem is obtained by replacing 'full' by 'empty', 'send' by 'receive', and 'outgoing' by 'incoming' in the previous problem.

Next, we give a theorem that shows how partial snapshots may be employed.

5.2 Termination Detection Theorem

Let v be a system computation such that Q terminates after v. If z is a system computation such that for all q in Q, $v.q$ is a prefix of $z.q$, then $v.q = z.q$, for all q in Q.

Proof We prove by induction on the length of prefixes u of z, that $u.q$ is a prefix of $v.q$, for all q in Q. In particular, we prove that $z.q$ is a prefix of $v.q$. Since $v.q$ is a prefix of $z.q$, it follows that $v.q = z.q$.

Base Case u is the empty sequence. The result holds, trivially.

Induction Step Consider a channel c used by processes q and r, where both q and r are in Q. Let $u.r = v.r$ (and $u.q$ is a prefix of $v.q$ from the induction hypothesis). From the monotonicity property, for all events

e on c, if the projection of (v, e) on c is not a computation of c, then the projection of (u, e) on c is not a computation of c. Since Q terminates after v, the projection of (v, e) on c is not a computation of c. Hence, if $u.r = v.r$, for all events e on c, the projection of (u, e) on c is not a computation of c. Since Q terminates after v, the only events on r after $v.r$ are events on channels to other processes in Q. Hence, if $u.r = v.r$, there is no event e on r such that (u, e) is a system computation.

From the arguments of the last paragraph, if (u, e) is a computation of z, then e is on a process r such that $u.r \neq v.r$. Since $u.r$ is a prefix of $v.r$, the length of $u.r$ is less than the length of $v.r$ In this case, $(u.r, e)$ is a prefix of $v.r$, since both $(u.r, e)$ and $v.r$ are prefixes of $z.r$, and the length of $(u.r, e)$ is at most the length of $v.r$.

5.3 Applications of the Theorem

The termination detection theorem tells us that old data $(v.q)$ is current (because $v.q = z.q$) if the old data shows that Q has terminated. This suggests the following class of algorithms for termination detection; this class includes algorithms in [1,4,9].

A Class of Algorithms for Termination Detection The algorithms employ a set of process computations $\{v.q | q \in Q\}$ and have the following specification.

Invariant $v.q$ is a prefix of $z.q$ where z is the system computation up to the current point.

Progress For all q, if the current value of $z.q$ is, say, $y.q$, then eventually $y.q$ is a prefix of $v.q$. (The progress property says that the process computations $v.q$ get updated: eventually, $v.q$ will include the current value, $y.q$, of $z.q$.)

The algorithm determines that Q has terminated if Q terminates after $\{v.q | q \in Q\}$, i.e., if Q terminates after a system computation, y, where $y.q = v.q$, all q in Q.

Correctness The proof of correctness of this class of algorithms is as follows. From the invariant and the theorem, if Q has terminated after $\{v.q|q \in Q\}$, then Q has terminated after z. From the progress property, if Q terminates after z, then eventually $v.q = z.q$, and hence eventually, the algorithm determines that Q has terminated.

Example Next, we give an example of algorithms with the invariant and progress properties given earlier. To detect termination of Q, a token is sent from process to process in Q, in such a manner that the token visits every process in Q repeatedly. The token carries with it a set of process computations $\{v.q|q \in Q\}$. Initially, $v.q$ is the empty sequence. When the token arrives at a process q, it updates this set, by replacing the value of $v.q$ in the set by its current computation, and q determines that Q has terminated if Q terminates after $\{v.q|q \in Q\}$.

Various optimizations are possible in applying this method to detect a specific form of termination. For example, to detect full-buffer deadlock, it is not necessary for the token to carry the entire computation $v.q$; it is sufficient for the token to contain the number of messages sent and received on each channel by q in $v.q$, and the set of processes for which q is waiting.

6 Summary

The paper presents necessary and sufficient conditions for a set of process computations to be a global snapshot. The condition is that for every channel, the computations in the snapshot of the processes that use the channel, are compatible with respect to the channel. The condition is helpful in the development of algorithms.

The paper also presents the concept of partial snapshots and demonstrates its utility.

References

[1] Chandy, K. M.[1987] 'A Theorem on Termination of Distributed Systems', TR-87-09, March 1987, Dept. of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712-1188.

[2] Chandy, K. M., and L. Lamport [1985]. 'Distributed Snapshots: Determining Global States of Distributed Systems,' ACM TOCS, 3:1, February 1985, pp. 63-75.

[3] Chandy, K. M. and J. Misra [1988] *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Massachusetts, 1988.

[4] Chandy, K. M. and J. Misra [1988] 'On Proofs of Distributed Algorithms with Application to the Problem of Termination Detection', submitted to Distributed Computing [1987].

[5] Dijkstra, E. W. [1985]. 'The Distributed Snapshot of K. M. Chandy and L. Lamport,' in Control Flow and Data Flow, ed. M. Broy, Berlin: Springer-Verlag, 1985, pp. 513-517.

[6] Fischer, M. J., N. D. Griffeth, and N. A. Lynch [1982]. 'Global States of a Distributed System,' IEEE Transactions on Software Engineering, SE-8:3, May 1982, pp. 198-202.

[7] Lamport,L.[1978] 'Time, Clocks and the Ordering of Events in a Distributed System,' C.ACM, 21:7, July 1978, pp 558-565.

[8] Hoare, C. A. R. [1984]. *Communicating Sequential Processes*, London: Prentice-Hall International, 1984.

[9] Raynal, M., J.-M. Helary, C. Jard, and N. Plouzeau [1987]. 'Detection of Stable Properties in Distributed Applications,' in Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, 1987, pp. 125-136.