Knowledge of the state of several, and possibly all, processes in a distributed system is often needed.

For example, a supervisory process must be able to detect when a subset of processes is deadlocked; a

process might migrate from one location to another or be replicated only after an agreement with others.

In all these examples a process needs to evaluate a predicate function of the global state of the system.

We call the process responsible for constructing the global state of the system the monitor. A monitor

sends messages requesting information about the local state of every process and gathers the replies to

construct the global state. Intuitively, the construction of the global state is equivalent to taking snapshots

of individual processes and then combining these snapshots into a global view.Yet, combining snapshots

is straightforward if and only if all processes have access to a global clock and the snapshots are taken

at the same time; hence, the snapshots are consistent with one another.

Arun is a total ordering R of all the events in the global history of a distributed computation consistent

with the local history of each participant process; a run

R =

e j1

1 , e j2

2 , . . . , e jn

n

(2.30)

implies a sequence of events as well as a sequence of global states.

For example, consider the three processes in Figure 2.8.We can construct a three-dimensional lattice

of global states following a procedure similar to the one in Figure 2.2, starting from the initial state

 (000) and proceeding to any reachable state  (i jk) with i , j , k the events in processes p1, p2, p3,

respectively. The run R1 =

e1

1, e1

2, e1

3, e2

1

is consistent with both the local history of each process and the global history; this run is valid, and the system has traversed the global states

 000, 100, 110, 111, 211. (2.31)

On the other hand, the run R2 = (e1

1, e2

1, e1

3, e3

1, e2

3) is invalid because it is inconsistent with the global

history. The system cannot ever reach the state  301; message m1 must be sent before it is received, so

event e1

2 must occur in any run before event e3

1.

A cut is a subset of the local history of all processes. If h j

i denotes the history of process pi up to

and including its j-th event, e j

i , then a cut C is an n-tuple

C = {h j

i

} with i ∈ {1, n} and j ∈ {1, ni }. (2.32)

The frontier of the cut is an n-tuple consisting of the last event of every process included in the cut.

Figure 2.8 illustrates a space-time diagram for a group of three processes, p1, p2, p3, and it shows two

cuts, C1 and C2. C1 has the frontier (4, 5, 2), frozen after the fourth event of process p1, the fifth event

of process p2, and the second event of process p3, and C2 has the frontier (5, 6, 3).

Cuts provide the necessary intuition to generate global states based on an exchange of messages

between a monitor and a group of processes. The cut represents the instance when requests to report

individual states are received by the members of the group. Clearly not all cuts are meaningful. For

example, the cut C1 with the frontier (4, 5, 2) in Figure 2.8 violates our intuition regarding causality; it

includes e4

2, the event triggered by the arrival of message m3 at process p2 but does not include e3

3, the

event triggered by process p3 sendingm3. In this snapshot p3 was frozen after its second event, e2

3, before

it had the chance to send messagem3. Causality is violated and the system cannot ever reach such a state.

Next we introduce the concepts of consistent and inconsistent cuts and runs. A cut closed under the

causal precedence relationship is called a consistent cut. C is a consistent cut if and only if for all events

∀e, e

, (e ∈ C) ∧ (e  → e) ⇒ e  ∈ C. (2.33)

A consistent cut establishes an “instance” of a distributed computation. Given a consistent cut we

can determine if an event e occurred before the cut.

A run R is said to be consistent if the total ordering of events imposed by the run is consistent with the

partial order imposed by the causal relation; for all events, e → e  implies that e appears before e  in R.

Consider a distributed computation consisting of a group of communicating processes

G = {p1, p2, . . . , pn}. The causal history of event e, γ (e), is the smallest consistent cut of G including

event e

γ (e) = {e  ∈ G|e  → e} ∪ {e}. (2.34)

The causal history of event e5

2 in Figure 2.9 is:

γ

e5

2

=

e1

1, e2

1, e3

1, e4

1, e5

1, e1

2, e2

2, e3

2, e4

2, e5

2, e1

3, e2

3, e3

3

. (2.35)

This is the smallest consistent cut including e5

2; indeed, if we omit e3

3, then the cut (5, 5, 2) would be

inconsistent; it would include e4

2, the communication event for receiving m3, but not e3

3, the sending of

m3. If we omit e5

1, the cut (4, 5, 3) would also be inconsistent and it would include e3

2 but not e5

1.

Causal histories can be used as clock values and satisfy the strong clock condition, provided that we

equate clock comparison with set inclusion. Indeed,

e → e  ≡ γ (e) ⊂ γ (e

). (2.36)

The following algorithm can be used to construct causal histories:

• Each pi ∈ G starts with θ = ∅.

• Every time pi receives a message m from p j it constructs

γ (ei ) = γ (e j ) ∪ γ (ek ) (2.37)

with ei the receive event, e j the previous local event of pi , ek the send event of process p j .

Unfortunately, this concatenation of histories is impractical because the causal histories grow very fast.

Now we present a protocol to construct consistent global states based on the monitoring concepts

discussed in this section. We assume a fully connected network; recall that given two processes pi and

p j , the state ξi , j of the channel from pi to p j consists of messages sent by pi but not yet received by

p j . The snapshot protocol of Chandy and Lamport consists of three steps [72]:

1. Process p0 sends to itself a “take snapshot” message.

2. Let p f be the process from which pi receives the “take snapshot” message for the first time. Upon

receiving the message, the process pi records its local state, σi , and relays the “take snapshot” along

all its outgoing channels without executing any events on behalf of its underlying computation.

Channel state ξ f ,i is set to empty, and process pi starts recording messages received over each of

its incoming channels.

3. Let ps be the process fromwhich pi receives the “take snapshot”message after the first time. Process

pi stops recording messages along the incoming channel from ps and declares channel state ξs,i as

those messages that have been recorded.

Each “take snapshot” message crosses each channel exactly once, and every process pi has made its

contribution to the global state. A process records its state the first time it receives a “take snapshot” message and then stops executing the underlying computation for some time. Thus, in a fully connected

network with n processes, the protocol requires n   (n − 1) messages, one on each channel.

For example, consider a set of six processes, each pair of processes being connected by two unidirectional

channels, as shown in Figure 2.10. Assume that all channels are empty, ξi , j = 0, i ∈ {0, 5},

j ∈ {0, 5}, at the time when process p0 issues the “take snapshot” message. The actual flow of messages

is:

• In step 0, p0 sends to itself the “take snapshot” message.

• In step 1, process p0 sends five “take snapshot” messages, labeled (1) in Figure 2.10.

• In step 2, each of the five processes p1, p2, p3, p4, and p5 sends a “take snapshot” message labeled

(2) to every other process.

A “take snapshot” message crosses each channel from process pi to p j , i , j ∈ {0, 5} exactly once

and 6   5 = 30 messages are exchanged.