Chapter 30

Computation of a Global Function

30.1 Introduction

As we have seen earlier, one of the fundamental difficulties of distributed computing is that no processor has access to the global state. This difficulty can be alleviated by developing mechanisms to compute functions of the global state. We call such functions *global functions*. More concretely, assume that we have x_i located at process P_i. Our aim is to compute a function $f(x_1, x_2, ..., x_n)$ which clearly depends on states of all processes.

First we present an algorithm for convergecast and broadcast on a network assuming that there is a pre-defined spanning tree. The convergecast requires information from all nodes of the tree to be collected at the root of the tree. Once all the information is present at the root node, it can compute the global function and then broadcast the value to all nodes in the tree. Both the convergecast and the broadcast requires a spanning tree on the network. We also give an algorithm to build the spanning tree.

Next we present an algorithm to compute a global function due to Bermond, Konig and Raynal. Their algorithm has many desirable properties. First, at the end of their algorithm all processors know the value of the function f. Moreover, the algorithm is completely symmetric. It does not assume that underlying topology of the communication network is known except that all processors know about their neighbors. The algorithm is presented first for a special case when the global function to be

computed is the routing table for each processor. Then, this algorithm is generalized so that it can be used for any global function.

We assume in this chapter that all links are bi-directional unless otherwise stated.

30.2 Convergecast and Broadcast

The algorithms for convergecast and broadcast are very simple if we assume a rooted spanning tree. For convergecast, each node in the spanning tree is responsible to report to its *parent* the information of its sub-tree. The variable *parent* for a node x is the identity of the neighbor of x which is the parent in the rooted spanning tree. For root, this value is null. The variable *numchildren* keeps track of the total number of its children and *numreports* keeps track of the number of its children who have reported. The algorithm is shown in Fig. 30.1. When the root node hears from all its children, it has all the information to compute the global function.

```
var
    parent: process id ;// initialized based on the rooted spanning tree
    numchildren: integer ; // initialized based on the rooted spanning tree
    numreports: integer initially 0;

on receiving a report from P_j
    numreports := numreports + 1;
    if (numreports = numchildren) then
        if (parent = null) then // root node
            compute global function;
        else send report to parent;
    endif;
```

Figure 30.1: A Convergecast Algorithm

The broadcast algorithm shown in Fig. 30.2 is dual of the convergecast algorithm. The algorithm is initiated by the root process by sending the broadcast message to all its children. In this algorithm messages traverse down the tree.

Thus, the only task remaining is to build a rooted spanning tree. We

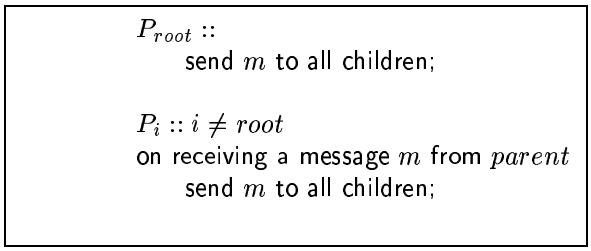


Figure 30.2: A Broadcast Algorithm

assume that there is a distinguished processor *root*. This algorithm, sometimes called the flooding algorithm, can also be used to broadcast a message m, when there is no predefined spanning tree. The algorithm for flooding a message is simple. Whenever a processor P_i receives a message m (from P_j) for the first time, it sends that message to all its neighbors except P_j. To P_j it sends a message indicating that P_j is the parent of P_i. The algorithm shown in Fig. 30.3 is initiated by the root processot by sending message m to all its neighbors. The proof of correctness is left as an exercise.

30.3 Global Functions

Before we present a general algorithm for computing a global function, we describe an algorithm which computes routing tables in a distributed system. This will not only make the general algorithm easier to understand, but also illustrate the application of the general algorithm.

30.3.1 An Algorithm to Compute the Routing Tables

The algorithm to compute the routing table in a distributed systems works in phases. Each process goes through initialization and then through a series of phases $phase_1, phase_2, ..., phase_n$. In each phase a process will possibly learn some new information. If a process does not receive any new information in any phase, then it terminates.

Let D be the diameter of the graph of the communication network topology. We assume that D is known. We use d to denote the number of the current phase, and c to denote any channel incident on any process P. All channels are assumed to be bi-directional.

The variables used by the algorithm are:

d : phase number.

inf : global information known by P
{ identities of the nodes for which P knows a shortest route }

new : new information obtained since the beginning of this phase.

sent(c) : message sent on channel c at the current phase.

received(c) : message received through channel c.

Rout(c) : set of processes for which channel c must be used by P.

The program for any process is in Figure 30.4.

Each process in the above algorithm first sends out some information on all channels incident to it. Then, it receives information along those channels. Based on that information, it updates its routing tables and the information that it will send out in the next phase.

It is easily seen that the above algorithm will generate correct routing tables.

30.3.2 A General Algorithm

We now generalize this algorithm for computing any global function. Further, we will not assume that D is known. We use the variable $OPEN$ to denote the set of channels open at any phase.

The algorithm is in Figure 30.5.

The above algorithm is synchronous — it works using the notion of phases. Synchronous algorithms are much easier to verify than asynchronous algorithms. The easiest method to verify these algorithms is to view them as sets of equations relating variables at phase d and $d-1$. The message sent on channel c in phase d is viewed as the state of the variable $sent_d(c)$. Using $\overline{c}$ to denote the channel at the other process, we also get that $received_d(c) = sent_d(\overline{c})$. Then, most claims can be verified using induction. On rewriting the algorithm, we obtain:

$$
\begin{aligned}
\forall c \in \text{open}_{d-1} : \text{sent}_d(c) &:= \text{new}_{d-1} - received_{d-1}(c) \\
\forall c \in \text{open}_{d-1} : \text{received}_d(c) &:= \text{sent}_d(\overline{c}) \\
\text{new}_d &:= \cup_c \text{ received}_d(c) - \text{inf}_{d-1} \\
\text{open}_d &:= \text{open}_{d-1} - \{\ c \mid \text{sent}_d(c) = \text{received}_d(c)\} \\
\text{inf}_d &:= \text{inf}_{d-1} \cup \text{new}_d
\end{aligned}
$$

Let $N^i(P)$ denote the information at nodes which are at distance i from P. When $i = -1$, $N^{-1}(P)$ will simply be $\emptyset$. Further, let

$$T^i(P) = \cup_{j \leq i} N^j(P).$$

The variable $T^i(P)$ denotes the information at nodes which are at distance less than or equal to i from P. The initial values for variables in the algorithm are

$inf_0 = N^0(P)$;
$new_0 = N^0(P)$;
$open_0 = all$;
$\forall c : \text{received}_0(c) = \emptyset$.

Before we present the properties of the algorithm, we need the following lemma.

Lemma 1 *Let P and Q be two adjacent processes.*

$$T^{d-1}(P) = T^{d-1}(Q)$$

if and only if

$$N^{d-1}(P) - N^{d-2}(Q) = N^{d-1}(Q) - N^{d-2}(P).$$

Proof: ($\Leftarrow$)
$T^{d-1}(P)$
$= T^{d-2}(Q) \cup N^{d-1}(P) \cup N^{d-2}(P)$ (nodes at distances $d-3$, $d-1$ and $d-2$)
$= T^{d-2}(Q) \cup \left(N^{d-1}(P) - N^{d-2}(Q)\right) \cup N^{d-2}(P)$ (because $N^{d-2}(Q) \subseteq T^{d-2}(Q)$)
$= T^{d-2}(Q) \cup \left(N^{d-1}(Q) - N^{d-2}(P)\right) \cup N^{d-2}(P)$ (given)
$= T^{d-2}(Q) \cup N^{d-1}(Q)$ ($N^{d-2}(P) \subseteq T^{d-1}(Q)$)
$= T^{d-1}(Q)$ (definition of T).

($\Rightarrow$)
Given $T^{d-1}(P) = T^{d-1}(Q)$, we have to show that $N^{d-1}(P) - N^{d-2}(Q) = N^{d-1}(Q) - N^{d-2}(P)$. By symmetry, it is sufficient to show $N^{d-1}(P) - N^{d-2}(Q) \subseteq N^{d-1}(Q) - N^{d-2}(P)$. Let $x \in N^{d-1}(P) - N^{d-2}(Q)$. This implies that x is at distance of $d-1$ from P and not at distance $d-2$ from Q. Its distance from Q cannot be smaller than $d-2$, otherwise its distance from P is strictly smaller than $d-1$. Since $T^{d-1}(P) = T^{d-1}(Q)$, it follows that x is exactly $d-1$ distance away from Q. Thus, $x \in N^{d-1}(Q) - N^{d-2}(P)$.

■

Now, we have the following theorem.

Theorem 1 *For all $d \geq 1$, the following holds.*

$$\begin{aligned} sent_d(c) &= N^{d-1}(P) - N^{d-2}(Q) \\ received_d(c) &= N^{d-1}(Q) - N^{d-2}(P) \\ new_d &= N^d(P) \\ open_d &= \{(P,Q) |\, T^{d-1}(P) \neq T^{d-1}(Q)\} \\ inf_d &= T^d(P) \end{aligned}$$

Proof: The proof is by induction on d. For $d = 1$, we get

$$\begin{aligned} \text{sent}_1(c) &= N^0(P) \\ \text{received}_1(c) &= N^0(Q) \\ \text{new}_1 &= N^1(P) \\ \text{open}_1 &= \{(P,Q) |\, T^{1-1}(P) \neq T^{1-1}(Q)\} = all \\ \text{inf}_1 &= T^1(P) \end{aligned}$$

These equations are easily verified by substituting the values at phase 0 in the program.

Now, we verify the general case when $d > 1$.

- sent_d
  $\text{sent}_d(c) = \text{new}_{d-1} - received_{d-1}(c)$ (from the program)
  $= N^{d-1}(P) - (N^{d-2}(Q) - N^{d-3}(P))$ (from induction)
  $= N^{d-1}(P) - N^{d-2}(Q)$; ($N^i(P)$ and $N^j(P)$ are disjoint for $i \neq j$)

- received_d
  $\text{received}_d(c) = \text{sent}_d(\overline{c})$ (from the program)
  $= N^{d-1}(Q) - N^{d-2}(P)$; (expression for sent)

- new_d
  $\text{new}_d = \cup\ \text{received}_d(c) - \text{inf}_{d-1}$ (from the program)
  $= \cup\ N^{d-1}(Q) - N^{d-2}(P) - T^{d-1}(P)$ (from induction hypothesis)
  $= N^d(P)$ (simplification)

- open_d
  $\text{open}_d = \text{open}_{d-1} - \{\ c \mid \text{sent}_d(c) = \text{received}_d(c)\}$ (from the program)
  $= \{(P,Q) |\ T^{d-2}(P) \neq T^{d-2}(Q)\} - \{\ c \mid N^{d-1}(P) - N^{d-2}(Q) = N^{d-1}(Q) - N^{d-2}(P)\}$ (from induction)
  $= \{(P,Q) |\, T^{d-1}(P) \neq T^{d-1}(Q)\}$ (simplification, Lemma 1)

- inf_d
  $\text{inf}_d = \text{inf}_{d-1} \cup \text{new}_d$ (from program)
  $= T^{d-1}(P) \cup N^d(P)$ (induction hypothesis)
  $= T^d(P)$.

■

The following lemma gives the message complexity of the above algorithm.

Lemma 2 *1. The number of messages is at most* $2(D+1)m$ *where* m *is the total number of channels.*
2. Assuming that the initial information is of size $\log n$*, the total number of bits communicated is at most* $O(mn \log n)$*.*
3. The running time of the algorithm is at most $(2D+1)\tau$ *where* τ *is the maximum transmission delay on a channel.*

Proof: 1. During each phase a process sends and receives one message along each open channel. There are D+1 phases in the algorithm.
2. Information about any node is transmitted on a channel at most twice.
3. A node will be woken up in at most $D\tau$ time. Each phase takes at most τ time and there are $D+1$ phases.

■

.

30.4 Problems

1. Analyze the time and message complexity of convergecast, broadcast and spanning tree construction algorithms.

2. Show that the spanning tree algorithm in Fig. 30.3 constructs a breadth-first search tree in a synchronous network (where every message takes one unit time and computation takes zero time). Also show that this is not true for an asynchronous network.

3. Give an algorithm that constructs a depth first search spanning tree assuming that there is a distinguished processor *root*.

4. Give an algorithm that constructs a depth first search spanning tree assuming that there is no distinguished processor. You may assume that each processor has unique processor id.

5. Consider a completely connected network of processors. What is the message complexity of algorithm in Fig. 30.5 to compute a global function?

6. Modify the algorithm compute a global function to incorporate the notion of open input and open output channels. Give conditions when a process P need not send further messages along a channel c. What is the message complexity of this algorithm?

7. How will you use the algorithm to compute a global function to compute (a) the diameter of a graph, (b) the centers of a graph, (c) the eccentricity of a graph?

30.5 Bibliographic Remarks

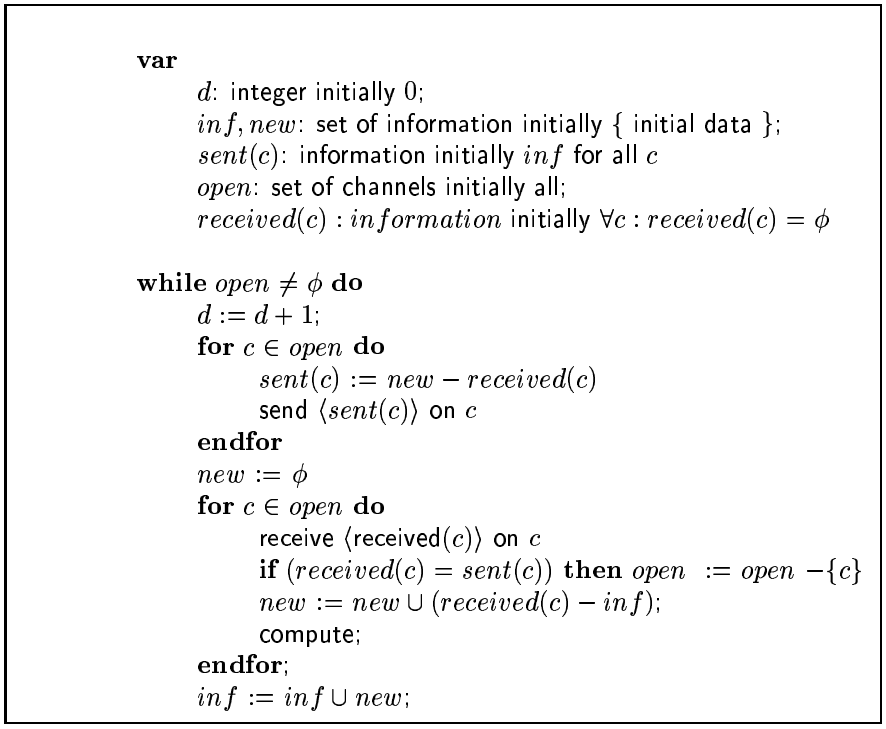
The algorithms for broadcast, convergecast and flooding appear to be folklore. The discussion of the algorithm for computation of a global function is taken from Bermond, Konig and Raynal[**?**].

```
var
    parent: process id initially null;
    numchildren: integer initially 0;
    childrenlist: list initially null;
    numneighbors: integer initially the number of neighbors;
    numreports: integer initially 0;
    firstflag: boolean initially true for all processes except the root;

Upon receiving a message m from P_j
    if firstflag then
        parent = P_j;
        firstflag := false;
        send m to all neighbors except P_j;
        send (parent) to P_j;
    else
        send (reject) to P_j;

Upon receiving a (parent) message
    numchildren := numchildren + 1;
    append(childrenlist, P_j);
    numreports := numreports + 1;
    if (numreports = numneighbors) then halt;

Upon receiving a (reject) message
    numreports := numreports + 1;
    if (numreports = numneighbors) then halt;
```

Figure 30.3: A Spanning Tree Construction Algorithm

```
var
    d: integer initially 0;
    inf: set of information initially { identity of the node };
    sent(c): information initially inf for all c

while d < D do
    d := d + 1;
    send⟨sent(c)⟩ on all channels c;
    new := φ;
    for every channel c do
        receive ⟨ received(c)⟩ on c;
        for y ∈ received(c) − inf − new do
            Rout (c) := Rout (c) ∪{y};
        new := new ∪ (received(c) − inf);
    endfor
    inf := inf ∪ new;
    sent(c) := new − received(c);
endwhile;
```

Figure 30.4: An Algorithm to Compute the Routing Tables.



```
var
    d: integer initially 0;
    inf, new: set of information initially { initial data };
    sent(c): information initially inf for all c
    open: set of channels initially all;
    received(c) : information initially ∀c : received(c) = φ

while open ≠ φ do
    d := d + 1;
    for c ∈ open do
        sent(c) := new − received(c)
        send ⟨sent(c)⟩ on c
    endfor
    new := φ
    for c ∈ open do
        receive ⟨received(c)⟩ on c
        if (received(c) = sent(c)) then open := open −{c}
        new := new ∪ (received(c) − inf);
        compute;
    endfor;
    inf := inf ∪ new;
```

Figure 30.5: An Algorithm to Compute a General Global Function

Chapter 31

Distributed Shared Memory

31.1 Introduction

A concurrent object is one which allows multiple processes to to execute its operations concurrently. For example, a concurrent queue in a shared memory system may allow multiple processes to invoke *enqueue* and *dequeue* operations. Or, in a client-server system, a server may handle concurrent remote procedure calls from multiple clients. The natural question, then, is to define which behavior of the object under concurrent operations is consistent (or correct). Consider the case when a process P first enqueues x and then dequeues while process Q concurrently enqueues y. Is the queue's behavior acceptable if process P gets y as the result of dequeue ? The objective of this chapter is to clarify such questions.

31.2 System Model

Objects and Processes.
A *concurrent system* consists of a finite set of *sequential processes* (named $p_1, p_2, ..., p_n$) that communicate through *concurrent objects*. Each object has a name and a type. The type defines the set of possible values for objects of this type and the set of primitives operations that provide the only means to manipulate objects of this type. Execution of an opera-

tion takes some time; this is modeled by two events, namely an *invocation* event and a *response* event. Let *op(arg,res)* be an operation on object X issued at p_i; *arg* and *res* denote *op*'s input and output parameters, respectively. Invocation and response events $inv(op(arg))$ *on* X *at* p_i and $resp(op(res))$ *from* X *at* p_i will be abbreviated as *inv(op)* and *resp(op)* when parameters, object name and process identity are not necessary. For any operation e, we use $proc(e)$ to denote the process and $object(e)$ to denote the object associated with the operation.

Histories.
An execution of a concurrent system is modeled by a poset $(H, <_H)$ where H is the set of operations and $<_H$ is an irreflexive transitive relation that captures "occurred before" relation between operations. The relation includes the following relations.

`Process Order`: $(proc(e) = proc(f) \wedge resp(e)$ occurred before $inv(f)$.

`Object Order`: $(object(e) = object(f) \wedge resp(e)$ occurred before $inv(f)$.

A process sub-history $\mathcal{H}|p_i$ ($\mathcal{H}$ at p_i) of a history $\mathcal{H}$ is the subposet of all events in $\mathcal{H}$ whose process names are p_i. An object sub-history is defined in a similar way for an object X; it denoted $\mathcal{H}|X$ ($\mathcal{H}$ at X). Two histories $\mathcal{H}$ and $\mathcal{H}'$ are *equivalent* if they are composed of exactly the same set of invocation and response events.

A history $(H, <_H)$ is a *sequential* history if $<_H$ is a total order. Such a history would happen if there were only one sequential process in the system. A sequential history is legal if it meets the sequential specification of all the objects. For example, if we are considering a read-write register x as a shared object, then a sequential history is legal if for every read operation that returns its value as v, there exists a write on that object with value v, and there does not exist another write operation on that object with different value between that write and the read.

31.3 Sequential Consistency

Definition 1 *A history $(H, <_H)$ is sequentially consistent if there exists a sequential history S equivalent to H such that S is legal and it satisfies process order.*

Thus, if a history is sequentially consistent if its execution is equivalent to a legal sequential execution and each process's behavior is identical in the concurrent and sequential execution. In the following histories, P, Q and R

are processes operating on shared registers x, y and z. We assumes that all registers have 0 initially. The response of a read operation is denoted by $ok(v)$ where v is the value returned, and the response of a write operation is denoted by $ok()$.

1. $H_1 = P\ write(x,1),\ Q\ read(x),\ Q\ ok(0),\ P\ ok()$
   H_1 is sequentially consistent because it is equivalent to the following legal sequential history.
   $Q\ read(x),\ Q\ ok(0),\ P\ write(x,1),\ P\ ok()$.

2. $H_2 = P\ write(x,1),\ P\ ok(),\ Q\ read(x),\ Q\ ok(0)$
   Somewhat surprisingly, H_2 is also sequentially consistent. Even though P got the response of its write before Q, it is okay for Q to have read an old value. Note that H_2 is a sequential history but not legal. However, it is equivalent to the following legal sequential history.
   $Q\ read(x),\ Q\ ok(0),\ P\ write(x,1) P\ ok()$.

3. $H_3 = P\ write(x,1),\ Q\ read(x),\ P\ ok(),\ Q\ ok(0),\ P\ read(x),\ P\ ok(0)$
   H_3 is not sequentially consistent. Any sequential history equivalent to H_3 must preserve process order. Thus, the read operation by P must come after the write operation. This implies that the read cannot return 0.

4. $H_4 = P\ write(x,1),\ P\ ok(),\ Q\ write(y,1),\ Q\ ok(),\ Q\ write(x,2),\ Q\ ok(), P\ write(y,2)$

Given a history, is it easy to check whether it is sequentially consistent? We now show that even for simple read-write registers, checking sequential consistency is NP-complete.

Theorem 2 *Given a concurrent history H on a read-write register, it is NP-complete to determine whether it is sequentially consistent.*

Proof: The problem is clearly in NP since the sequential history S serves as a witness for sequential consistency. The NP-hardness follows using a reduction from a 3-SAT problem. The details of the reduction are left as an exercise.

■

Since sequential consistency is NP-complete, it is natural to impose additional constraints on the system. Two of the constraints commonly used are:

- *Write-write constraint*: We assume that the system totally orders all write operations on all objects.

- *Object-ordered constraint*: We assume that the system totally orders all pairs of conflicting operations ((read, write) and (write, write)) on every object.

We now have the following polynomial time algorithms. We first extend the definition of legality to any history (not just sequential history). A history $(H, <_H)$ is legal if for every read operation $r_i(x)$ that returns value as v

- There exists a write $w_j(x)v$
- There does not exist another write operation object with different value between $w_j(x)$ and the read $r_i(x)$. That is, there does not exist $w_k(x)u$ such that $(w_j(x)v <_H w_k(x)u) \quad \wedge \quad (w_k(x)u <_H r_i(x)v)$.

Theorem 3 *Let H be a history under WW-constraint. If H is legal then it is sequentially consistent.*

Proof: Assume that H is legal. Consider the acyclic graph corresponding to H. This graph is acyclic and legal. We will add edges to this graph one at a a time preserving acylicity and legality until all operations on a single object are ordered.

We proceed as follows. Given any read operation $r_i(x)v$ from write $w_j(x)v$, and any write operation $w_k(x)u$ such that $w_j(x)$ is ordered before $w_k(x)$, we put an edge from $r_i(x)$ to $w_k(x)$. This cannot create a cycle, because legality of H implies that there is no path from $w_k(x)$ to $r_i(x)$. Further, we preserve legality of the graph as shown next.

Legality can only be violated if addition of an edge results in creation of a path from some write operation, say w_a, to some read operation, say r_b. Assume, if possible, that by adding the edge from r_i to w_k, a new path from w_a to r_b is formed. Thus, there was a path before from w_a to r_i and a path from w_k to r_b. Since writes are totally ordered, we have:
case 1: $w_a \leq w_k$. But, this implies that there is a path from w_a to r_b which does not use the edge (r_i, w_k). This path goes from w_a to w_k and then from w_k to r_b.
case 2: $w_k < w_a$. But this implies that there was a path from w_k to w_a in the original graph.

Thus, by adding this edge we have preserved acylicity and legality. We continue to do so until every read operation on any object x is ordered with respect to all write operations. Now consider S any linear extension of the resulting graph. It is easy to see that S preserves process order as well as legality.

■

Theorem 4 *Let H be a history under OO-constraint. If H is legal, then it is sequentially consistent.*

Proof: Assume that H is legal. Consider any linear order S compatible with H. It is easy to verify that S is also legal.

■

The above results indicate that so long as we can enforce legality of reads under WW or OO-constraint, we are guaranteed to have a sequentially consistent execution. We will now restrict our attention to WW-constraint. Under WW-constraint, we can assume that all writes are totally ordered by the system. The total order at any point of time is called the ww-list. It is necessary and sufficient to ensure that every read is from the most recent write. That is, if a read operation on x reads from w_j then there does not exist a write which was after w_j and causally preceded r_i. The necessity of reading from the most recent write is obvious due to legality. It is sufficient, because future additional relations cannot make this read illegal. That is, no later write can have a path to this read. This is because an edge from operation s to t implies that s occurred before t in real-time.

31.3.1 Local Read Algorithm

Assume that every processor's memory is as big as the shared memory.

```
P_i::
        Upon read(x):
            generate Ok(v) where v is the value of P_i's copy of x;

        Upon write(x, v):
            atomic broadcast send (x, v);

        Upon receive of atomic-broadcast from P_j:
            set local copy of x to v
            if i = j then generate Ok() for write;
```

Figure 31.1: Sequentially consistency: local read algorithm

This algorithm follows WW-constraint since all writes are ordered. The equivalent sequential history can be constructed as follows. All writes are first arranged in the order of atomic broadcast. Now all reads are inserted