Distributed Systems HW2

Samuel Alaniz, Juan Neri

Fall 2022

1 Show that $G \cap H$ and $G \cup H$ are consistent cuts

1.1 $G \cap H$

Since both G and H are consistent global cuts, by definition they both satisfy the following:

- $g \in G \wedge g' \rightarrow g \implies g' \in G$
- $h \in H \wedge h' \rightarrow h \implies h' \in H$

Now let $i \in (G \cap H) \wedge i' \rightarrow i$. By definition of intersection, then $i \in G$ and $i \in H$ and therefore (by the above two items) $i' \in G$ and $i' \in H$. Again by the definition of intersection this means that $i' \in (G \cap H)$

$\therefore i \in (G \cap H) \wedge i' \rightarrow i \implies i' \in (G \cap H)$

1.2 $G \cup H$

Since both G and H are consistent global cuts, by definition they both satisfy the following:

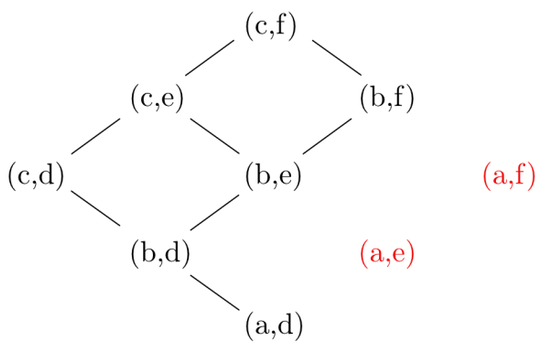
- $g \in G \wedge g' \rightarrow g \implies g' \in G$
- $h \in H \wedge h' \rightarrow h \implies h' \in H$

Now let $i \in (G \cup H) \wedge i' \rightarrow i$. By definition of union, then $i \in G$ or $i \in H$ and therefore (by the above two items) $i' \in G$ or $i' \in H$. Again by the definition of union this means that $i' \in (G \cup H)$

$\therefore i \in (G \cup H) \wedge i' \rightarrow i \implies i' \in (G \cup H)$

2 Show all consistent cuts using a Hasse-diagram

From all consistent cuts in this computation, only (a, e) and (a, f) are not consistent. This is because in both cases $e \in F \wedge b \rightarrow e$ but $b \notin F$



3 Snapshot algorithm

3.1 Show a possible global state of the computation in which channels c and d have at least one message each

- $(x = 6, c = 7, y = 4, d = 5)$

3.2 What are the values of x and y at the termination of the program?

- $x = 5$
- $y = 14$

3.3 Show a sequence of events that can lead to this snapshot.

1. P sets $x = 7$
2. P sends 7 via c
3. P sets $x = 6$
4. Q sets $y = 4$
5. Q sends 5 via d
6. P receives 5 via d and sets $x = 5$

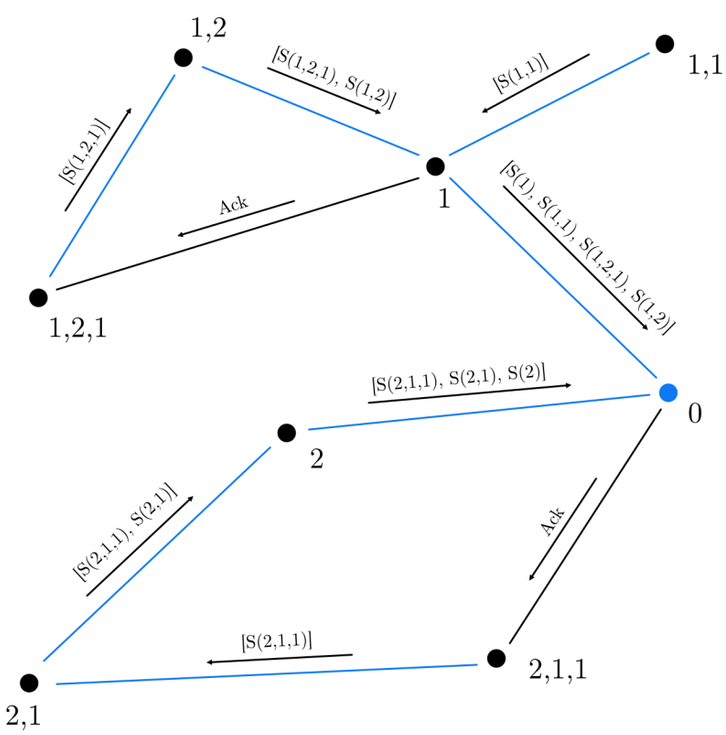
4 Extend Chandy and Lamport

The main idea is to find a spanning tree with root at P_0 after each process has finished collecting its state. We propagate the local state of each process (starting from the leafs of the tree), accumulating the values of each subtree until they reach the root P_0 at which point the algorithm is complete.

Start by performing the original Chandy and Lamport algorithm. Once all P_0 incoming channels are closed, P_0 proceeds to request the local state of each of its neighbor process (children) 1 by 1 only proceeding to P_i after it has heard back from P_{i-1} and confirmed that the incoming message did not already include the state of P_i. In turn, each child acts in the same way, responding to its parent only once it has accumulated the responses from each of its own children. To prevent loops, if a process that is waiting for its children responses receives a request, it will respond with an ack to signify that it has alrady been added to the tree.

In the diagram below, P_0 is represented by the blue vertex 0 and the resulting spanning tree is represented by the blue edges. In addition $S(i)$ represents the state of process i as captured by Chandy and Lamport Global Snapshot Algorithm. We can see that each process sends a list with the state of its entire sub-tree to its parent, forming a complete global snapshot at the root P_0

The original Chandy-Lamport algorithm requires that a marker be sent along all channels. Our addition requires that another set of requests and responses be sent on each channel. Therefore the message complexity of this algorithm is $3m$ (O(m)) where m is the number of channels in the network.



5 Leader election on a chain of processes

- To initiate the program, any process needs to send an 'init' message to its left neighbor, and gets propagated until it reaches the leftmost process (the leftmost process can skip this step).
- Once the leftmost process gets the 'init' message, it begins the leader election by sending an 'election' message with its own pid to its right neighbor.
- Any process that gets an 'election' message from its left neighbor, either forwards it to its right neighbor if its id is lower than the incoming request, otherwise its sends its own id.
- Once the 'election' message reaches the rightmost process, it is known that leader id will be the largest between the incoming id and its own id.
- Finally the rightmost process can proceed to forward the leader id by sending a 'leader' message to its left neighbor . Each receiving process can update their knowledge of the leader id, and forward the message until it reaches the left most process

P_i ::
var
$myid$: integer;
$leaderid$: integer initially null;
$leftmost$: boolean initially true if no left neighbor;
$rightmost$: boolean initially true if no right neighbor;

To initiate election:
if $leftmost$ **:**
send $(election, myid)$ to right neighbor
else:
send $(init)$ to left neighbor

Upon $receive(init)$ from right neighbor
if $leftmost$ **:**
send$(election, myid)$ to right neighbor
else:
send$(init)$ to left neighbor

Upon $receive(election, otherid)$ from left neighbor
if $rightmost$ **:**
if $otherid < myid$ **:**
$leaderid = myid$
else:
$leaderid = otherid$
send$(leader, leaderid)$ to left neighbor

Upon $receive(leader, leaderid)$ from right neighbor
$leaderid = leaderid$
if ! $leftmost$ **:**
send$(leader, leaderid)$ to left neighbor

Figure 1: Juan and Sammy Leader Election Algorithm

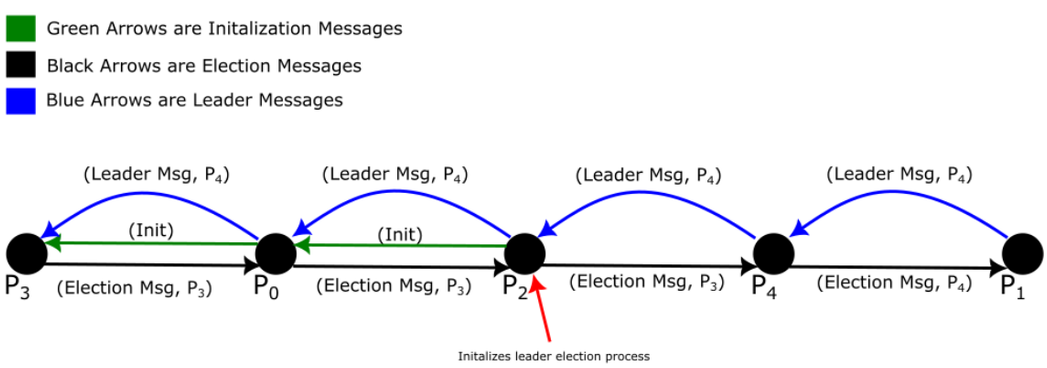


Figure 2: Visualization of Juan and Sammy Leader Election Algorithm