Notes on Agreement Protocols
Source: Prepared by Prof. Neeraj Mittal of CS Department, U.T.Dallas

An agreement problem typically requires all "non-faulty" processes to come to an agreement or consensus. A process is *nonfaulty* (or *correct*) if it always behaves according to the specifications; otherwise it is *faulty*. Once a process fails, it may start behaving arbitrarily and even maliciously. It can lie (*byzantine failure*), omit messages selectively (*omission failure*) or stop completely (*crash failure*). In this course, we will study three variants of the agreement problem, namely *byzantine agreement*, *consensus*, and *interactive consistency*. In all three agreement problems, one or more processes propose a value and all correct (or nonfaulty) processes must agree on either a value or a vector of values. An agreement protocol is typically described in terms of two primitives, namely `propose( )`, and `decide( )`. To avoid confusion, we will use v to represent the value proposed and u to represent the value decided. Also, let n denotes the number of processes. Unless otherwise stated, assume that the communication topology is fully connected. We now describe the three agreement problems.

1 Agreement Problems

1.1 Byzantine Agreement Problem

In the byzantine agreement problem, there is a distinguished process called the *source process*, which we represent by P_s. Only the source process can propose a value and it does so by calling `BA_propose(`v_s`)`. A non-source process does not propose a value and simply calls `BA_propose( )`. A process decides on a value by calling `BA_decide( )`. Specifically, process P_i decides on a value u_i by invoking `BA_decide( )`. A protocol that solves the byzantine agreement problem must satisfy the following properties:

Termination: All nonfaulty processes should eventually decide. In other words, `BA_decide( )` should eventually terminate for all nonfaulty processes.

Agreement: All nonfaulty processes decide on the same value. Formally,

$$\langle \forall i, j : P_i \text{ and } P_j \text{ are nonfaulty} : u_i = u_j \rangle$$

Validity: If the source process is nonfaulty, then all nonfaulty processes decide on the value proposed by the source process. Formally,

$$P_s \text{ is nonfaulty} \;\Rightarrow\; \langle \forall i : P_i \text{ is nonfaulty} : u_i = v_s \rangle$$

1.2 Consensus Problem

In the consensus problem, all processes propose a value. Specifically, process P_i proposes value v_i by executing `Cons_propose`(v_i) and decides on a value u_i by calling `Cons_decide( )`. A protocol that solves the consensus problem must satisfy the following properties:

Termination: All nonfaulty processes should eventually decide. In other words, `Cons_decide( )` should eventually terminate for all nonfaulty processes.

Agreement: All nonfaulty processes decide on the same value. Formally,

$$\langle \forall i, j : P_i \text{ and } P_j \text{ are nonfaulty} : u_i = u_j \rangle$$

Validity: If all nonfaulty processes propose the same value, then all of them decide on the value proposed. Formally,

$$\langle \forall i, j : P_i \text{ and } P_j \text{ are nonfaulty} : v_i = v_j \rangle \;\Rightarrow\; \langle \forall i : P_i \text{ is nonfaulty} : u_i = v_i \rangle$$

1.3 Interactive Consistency Problem

In the interactive consistency problem, all processes propose a value. Specifically, process P_i proposes value v_i by executing `IC_propose`(v_i), and, unlike in the other two problems, decides on a vector of values $\langle u_{i1}, u_{i2}, \ldots, u_{in} \rangle$, instead of a single value, by calling `IC_decide( )`. A protocol that solves the interactive consistency problem must satisfy the following properties:

Termination: All nonfaulty processes should eventually decide. In other words, `IC_decide( )` should eventually terminate for all nonfaulty processes.

Agreement: All nonfaulty processes decide on the same vector. Formally,

$$\langle \forall i, j : P_i \text{ and } P_j \text{ are nonfaulty} : \langle \forall k : u_{ik} = u_{jk} \rangle \rangle$$

Validity: If P_i is nonfaulty, then the i^{th} entry in the vector of all nonfaulty processes is v_i. Formally,

$$P_i \text{ is nonfaulty} \;\Rightarrow\; \langle \forall j : P_j \text{ is nonfaulty} : u_{ji} = v_i \rangle$$

2 Impossibilty of Solving Agreement Problems in General

Fischer, Lynch and Paterson in their seminal paper proved that solving an agreement problem in a completely asynchronous system is impossible even if at most one process can fail by crashing. An interesting question then arises is: *under what assumption can we solve the agreement problem?* It turns out that the agreement problem can be solved if we assume that the system is synchronous, that is, processes run in lock step manner. A step of a synchronous system is referred to as a *round*. In every round, each process can (1) send message to one or more processes, (2) receive the messages sent to it by other processes in that round, and (3) perform some local computation. In addition to assumption that the system is synchronous, we assume the following:

- The topology is fully connected, that is, every process is connected to every other process.
- The channels are reliable. However processes are not reliable and are prone to failures.

- A process receiving a message always knows the identity of the process that sent the message.

We now prove that, in a synchronous distributed system, all three agreement problems are equivalent in the sense that a protocol that solves one of the problems can be easily transformed to solve the other two. The equivalence result does not depend on the specific failure model.

3 Equivalence of the Agreement Problems when the System is Synchronous

Interactive Consistency Protocol for process P_i:

```
IC_propose(v_i)
{
    // start n instances of the byzantine
    // agreement protocol

    // P_i is the source process for
    // the i^th instance

    for each j in [1,n] do
        if (i = j) then
            BA_propose(i, v_i);
        else
            BA_propose(j);
        endif;
    endfor;
}
```

```
IC_decide( )
{
    // j^th entry of the vector is the value
    // decided by the j^th instance of the
    // byzantine agreement protocol

    for each j in [1,n] do
        u_ij := BA_decide(j);
    endfor;
    return ⟨u_i1, u_i2, ..., u_in⟩;
}
```

Figure 1: Solving Interactive Consistency using Byzantine Agreement.

Sometimes, in order to solve an agreement problem $\mathcal{A}$ using a solution to another agreement problem $\mathcal{B}$, we are required to use multiple instances of the solution to $\mathcal{B}$. To distinguish between these various instances, whenever necessary, we assume that the propose and decide primitives take an additional argument that indicates the instance number. We first describe a protocol for solving the interactive consistency problem using a byzantine agreement protocol. The protocol is shown in Figure 1 and is self-explanatory. To prove the correctness of the protocol, we need to prove that the protocol satisfies the three required properties.

Termination: It follows from the fact that each instance of byzantine agreement protocol will eventually terminate.

Agreement: Each nonfaulty process obtains the j^{th} entry of its vector using the j^{th} instance of the byzantine agreement protocol. Thus, from the agreement property of the byzantine agreement protocol, it follows that, for each j, the j^{th} entry of the vector will be identical for all nonfaulty processes.

Validity: Clearly, if process P_j is nonfaulty, the j^{th} instance of the byzantine agreement protocol will decide on v_j—the value proposed by P_j.

We next describe a protocol for solving the consensus problem using an interactive consistency protocol. Note that our solution assumes that a *majority of processes are nonfaulty*. The protocol is shown in Figure 2.

Consensus Protocol for process P_i:

```
Cons_propose(v_i)
{
    IC_propose(v_i);
}

Cons_decide( )
{
    ⟨u_i1, u_i2, ..., u_in⟩ := IC_decide( );
    // return the value that occurs in
    // (absolute) majority
    // if there is no absolute majority,
    // then return the default value
    return majority{u_i1, u_i2, ..., u_in};
}
```

Figure 2: Solving Consensus using Interactive Consistency.

We now prove the correctness of the consensus protocol.

Termination: It follows from the fact that the interactive consistency protocol will eventually terminate.

Agreement: Clearly, all nonfaulty processes agree on all entries of their vectors. Therefore, the majority function will evaluate to the same value for all nonfaulty processes.

Validity: The validity property of the interactive consistency protocol guarantees that if process P_i is nonfaulty, then all nonfaulty processes will decide on v_i as the i^{th} component of their respective vectors. Now, suppose all nonfaulty processes propose the same value, say v. Since a majority of processes are nonfaulty, a majority of entries in the vector for all nonfaulty processes will be v. Therefore, all nonfaulty processes, in the consensus protocol, a majority of entries in the vector will be v.

A protocol for solving the byzantine agreement problem using a consensus protocol is presented in Figure 3. The proof of correctness of this protocol and the next three protocols is left as an exercise. A protocol for solving the byzantine agreement problem using an interactive consistency protocol is described in Figure 4. A protocol for solving the interactive consistency problem using a consensus protocol is shown in Figure 5. Finally, a protocol for solving the consensus problem using a byzantine agreement protocol is shown in Figure 6. Again, our solution assumes that a *majority of processes are nonfaulty.*

4 An Algorithm for Solving the Byzantine Agreement Problem

We investigate the byzantine agreement problem under the *byzantine failure model.* Let m denote the maximum number of faulty processes. It can be proved that no protocol exists for solving

```
Byzantine Agreement Protocol:

// propose primitive for the source process P_s
BA_propose(v_s)
{
    broadcast v_s to all processes;
    Cons_propose(v_s);
}

// propose primitive for process P_i, i ≠ s
BA_propose( )
{
    let v_i be the value received from process P_s;
    // if no message is received, then set v_i to
    // a default value
    Cons_propose(v_i);
}

// decide primitive for process P_i
// (P_i could be P_s)
BA_decide( )
{
    u_i := Cons_decide( );
    return u_i;
}
```

Figure 3: Solving Byzantine Agreement using Consensus.

```
Byzantine Agreement Protocol:

// propose primitive for the source process P_s
BA_propose(v_s)
{
    IC_propose(v_s);
}

// propose primitive for process P_i, i ≠ s
BA_propose( )
{
    IC_propose(0);
}

// decide primitive for process P_i
// (P_i could be P_s)
BA_decide( )
{
    ⟨u_i1, u_i2, ..., u_in⟩ := IC_decide( );
    return u_is;
}
```

Figure 4: Solving Byzantine Agreement using Interactive Consistency.

the byzantine agreement problem if $n \leqslant 3m$. To understand why this assumption is necessary, consider the following three scenarios assuming $n = 3$ and $m = 1$. In all three scenarios, P_1 is the source process. The faulty process is shown in oval; nonfaulty processes are shown in circles. Since nonfaulty processes do not know whether the source process is faulty, nonsource processes need to exchange values for at least two rounds. The argument assumes that nonsource processes exchange values for only two rounds. The argument, however, can be extended to any number of rounds.

- *Scenario 1:* P_1 is nonfaulty and proposes 0. P_3 is faulty.
  - *Round 1:* P_1 sends 0 to P_2 and P_3.
  - *Round 2:* P_2 sends 0 to P_3 and P_3 sends 1 to P_2.
- *Scenario 2:* P_1 is faulty.

```
Interactive Consistency Protocol for process P_i:

IC_propose(v_i)                                  IC_decide( )
{                                                {
    broadcast v_i to all processes;                  // j^th entry of the vector is the value
    let v_ij be the value received from              // decided by the j^th instance of the
        process P_j;                                 // consensus protocol
    // if no message is received from P_j,           for each j in [1, n] do
    // then set v_ij to a default value                  u_ij := Cons_decide(j);
                                                     endfor;
    // start n instances of the consensus            return ⟨u_i1, u_i2, ..., u_in⟩;
    // protocol                                  }
    for each j in [1, n] do
        Cons_propose(j, v_ij);
    endfor;
}
```

Figure 5: Solving Interactive Consistency using Consensus.

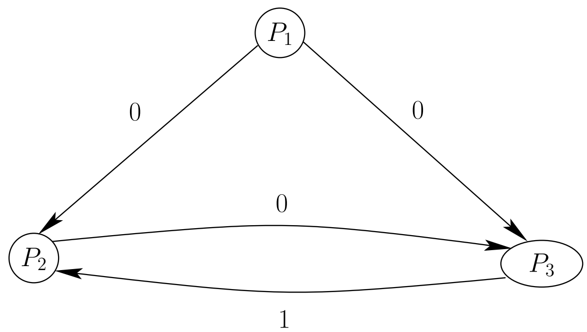
```
Consensus Protocol for process P_i:

Cons_propose(v_i)                                Cons_decide( )
{                                                {
    // start n instances of the byzantine            for each j in [1, n] do
    // agreement protocol                                u_ij := BA_decide(j);
                                                     endfor;
    // P_i is the source process for                 // return the value that occurs in
    // the i^th instance                             // (absolute) majority
                                                     // if there is no absolute majority,
    for each j in [1, n] do                          // then return the default value
        if (i = j) then                              return majority{u_i1, u_i2, ..., u_in};
            BA_propose(i, v_i);                  }
        else
            BA_propose(j);
        endif;
    endfor;
}
```

Figure 6: Solving Consensus using Byzantine Agreement.

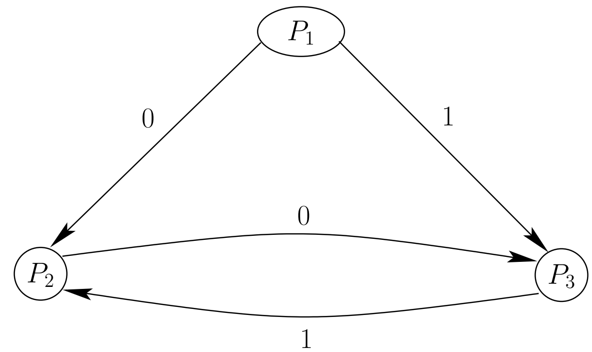
  - *Round 1:* P_1 sends 0 to P_2 and 1 to P_3.
  - *Round 2:* P_2 sends 0 to P_3 and P_3 sends 1 to P_2.

- *Scenario 3:* P_1 is nonfaulty and proposes 1. P_2 is faulty.
  - *Round 1:* P_1 sends 1 to P_2 and P_3.
  - *Round 2:* P_2 sends 0 to P_3 and P_3 sends 1 to P_2.

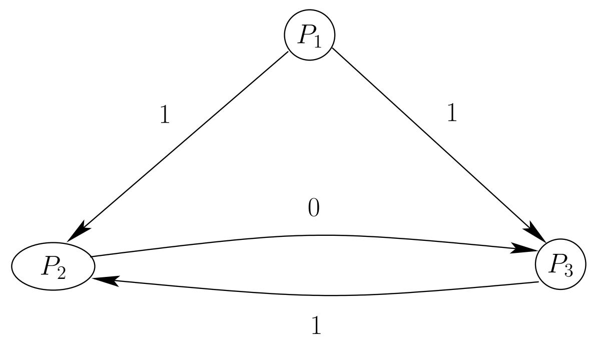
Observe that, due to the validity propoerty, P_2 has to eventually decide on 0 in Scenario 1 (P_1



(a) Scenario 1: P_3 is faulty.



(b) Scenario 2: P_1 is faulty.



(c) Scenario 3: P_2 is faulty.

Figure 7: Three different scenarios.

is nonfaulty and is proposing 0). Also, observe that P_2's view is identical in the two scenarios: Scenario 1 and Scenario 2. Specifically, it receives 0 from P_1 in the first round and 1 from P_3 in the second round in both the scenarios. Therefore, P_2 should decide on 0 in Scenario 2 as well. Likewise, using Scenario 3, we can argue that P_3 should decide on 1 in Scenario 2. We now have a violation of the agreement property in Scenario 2: P_2 eventually decides on 0 and P_3 eventually decides on 1.

Therefore to solve the byzantine agreement problem, we assume that $n \geqslant 3m+1$. The algorithm given here was developed by Lamport, Shostak and Pease. However, the description and the proof of correctness are based on the book by Nancy Lynch titled *Distributed Algorithms*.

4.1 Lamport, Shostak and Pease's Algorithm

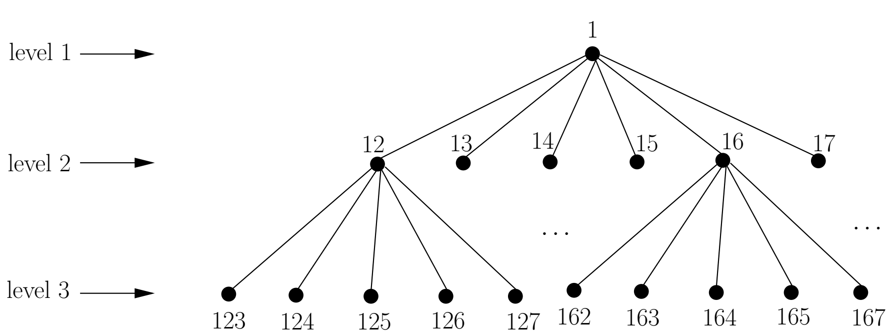


Figure 8: An Exponential Information Gathering Tree.

```
Lamport, Shostak and Pease algorithm for the source process P_s:

BA_propose(v_s)
{
    send v_s to all processes;
}

BA_decide( )
{
    return v_s;
}
```

Figure 9: Steps for the source process.

The algorithm belongs to the class of *Exponential Information Gathering* algorithms. The algorithm consists of two phases. In the first phase of the algorithm, which involves gathering information from other process, each nonsource process constructs a tree. The r^{th} level of the tree is constructed in the r^{th} round. There is a unique label associated with every node. A node at level r has a label that consists of r *distinct* process indices. As a process constructs the r^{th} level of the tree, it assigns a value to each node (or label) at that level. If the label $i_1 i_2 \cdots i_{j-1} i_j$, where $i_1 = s$, has the value v at process P_i, then it means that "P_{i_j} told P_i in round j that $P_{i_j - 1}$ told P_{i_j} in round $j-1 \cdots P_{i_1}$ told P_{i_2} in round 1 that its value is v". Figure 8 depicts the first three levels of the exponential information gathering tree. The first phase of the algorithm terminates after $m+1$ rounds. In the second phase of the algorithm, the tree that has been constructed is folded. Specifically, starting from leaf nodes, each process computes a new value for a node, using the majority rule, until the root node is reached. Lamport, Shostak and Pease's algorithm for solving the byzantine agreement problem for the source process P_s is shown in Figure 9. Lamport, Shostak and Pease's algorithm for a nonsource process is presented in Figure 10.

We next show a sample execution of the algorithm for $n = 4$ and $m = 1$. In our example, we assume that the source process is P_1 and that P_1 is faulty. As shown in Figure 11(a), in the first round, P_1 sends 1 to P_2 but 0 to P_3 and P_4. After the end of first round, all three nonsource processes P_2, P_3 and P_4 construct the first level of the tree as shown in Figure 11(b). The label of the root node in all the three trees is 1.

The second round of the algorithm is depicted in Figure 12(a). Each nonsource process adds the second level to the tree as shown in Figure 12(b).

Finally, Figure 13 depicts each tree after new values for all nodes (or labels) have been computed. The new value is shown in square box.

We now prove the correctness of the algorithm. In our proofs, we refer to $val(x)_i$ as the "old" value for label x at process P_i. Likewise, we refer to $newval(x)_i$ as the "new" value for x at P_i.

Lemma 1 *After $m+1$ rounds, all nonfaulty processes agree on the old value for every label that ends with the index of a nonfaulty process. Formally, for all nonfaulty processes P_i and P_j,*

$$x \text{ ends with index of a nonfaulty process} \;\Rightarrow\; val(x)_i = val(x)_j$$

Proof: Let x be of the form yk, where P_k is a nonfaulty process, and let $|y| = r$. Note that P_k

Lamport, Shostak and Pease algorithm for a nonsource process P_i, $i \neq s$:

```
BA_propose( )
{
    // round 1
    let v_i be the value received from P_s;
    val(s)_i = v_i;
    // if no value is received, then set val(s)_i to the default value

    // round r, 2 ⩽ r ⩽ m + 1
    for r = 2 to m + 1 do
        send all pairs (x, val(x)_i) such that |x| = r − 1 and x does not contain i
            to all nonsource processes;
        // construct the r^th level of the tree
        for each label xj such that |x| = r − 1 and x does not contain j do
            if a message arrives from P_j saying that the value of x is v then
                val(xj)_i = v;
            endif;
            // if no such message is received, then set val(xj)_i to the default value
        endfor;
    endfor;
}
```

```
BA_decide( )
{
    for each leaf node x do
        newval(x)_i = val(x)_i;
    endfor;

    for r = m to 1 do
        for each node x at level r do
            // find the absolute majority among the children of x
            newval(x)_i = majority_{j∉x}{ newval(xj)_i };
            // if there is no absolute majority, then set newval(x)_i to the default value
        endfor;
    endfor;

    return newval(s)_i;
}
```

Figure 10: Steps for a nonsource process.

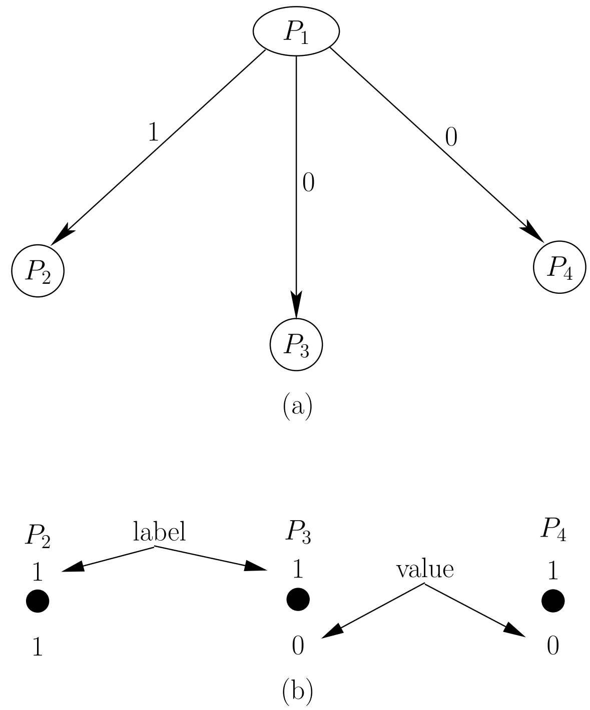


Figure 11: (a) The first round of the algorithm, and (b) the information gathering trees at the three nonsource processes after the end of first round.

sends the same value, say v, for y to both P_i and P_j in round $r+1$. Since both P_i and P_j are nonfaulty, both set the value of yk to v. Hence $val(x)_i = val(x)_j$. $\square$

Lemma 2 *After $m+1$ rounds, all nonfaulty processes agree on the new value for every label that ends with the index of a nonfaulty process. Moreover, the new value for such a label is same as its old value. Formally, for every label x that ends with the index of a nonfaulty process, there exists a value v, such that:*

$$P_i \textit{ is nonfaulty} \;\;\Rightarrow\;\; newval(x)_i = val(x)_i = v$$

Proof: Consider a label x that ends with the index of a nonfaulty process. It follows from Lemma 1 that all nonfaulty processes agree on the old value for x. Let this "common" old value for x be v. We now show that every nonfaulty process sets the new value for x to v as well. Consider a nonfaulty process P_i. The proof is by induction on the length of x.

Base Case [$|x| = m+1$]: In this case, x is a leaf node. Clearly, from the algorithm, $newval(x)_i = val(x)_i = v$.

Induction Step [$|x| = r$, where $1 \leqslant r \leqslant m$]: Consider a child of x with label xk, where P_k is a nonfaulty process. Since $val(x)_k = v$, P_k sends v as the value for x to all processes in round $r+1$. Since P_i is a nonfaulty process, on receiving this message, P_i sets the value for xk to v, that is, $val(xk)_i = v$. From induction hypothesis, $newval(xk)_i = val(xk)_i$. Therefore $newval(xk)_i = v$. In other words, all children of x whose label ends with the index of a nonfaulty process set v as the new value for their label. Now, x has $n-r$ children, which is at least $2m+1$. Note that the label of at most m of these children can end with the index of a faulty process. Therefore the label of a

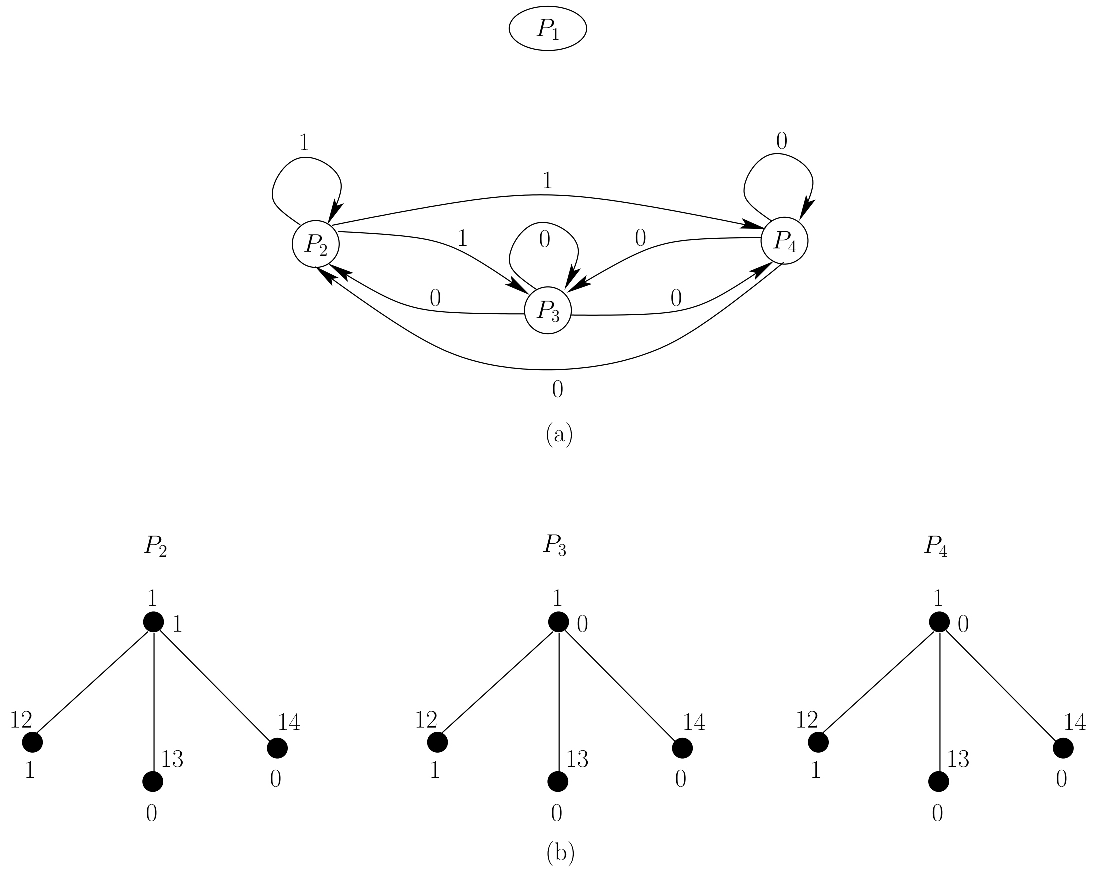


Figure 12: (a) The second round of the algorithm, and (b) the information gathering trees at the three nonsource processes after the end of second round.

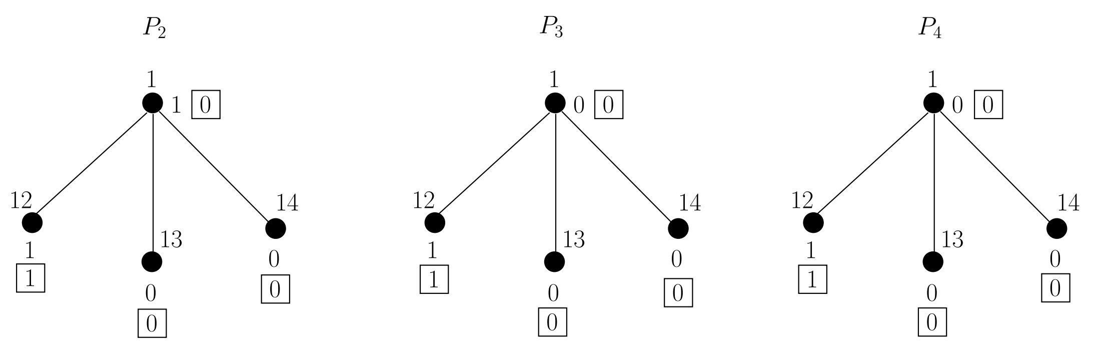


Figure 13: Each tree after the computation of the new value for all nodes.

majority of the children of x ends with the index of a nonfaulty process. Thus, from the algorithm, $newval(x)_i = v$. □

Observe that Lemma 2 establishes that the protocol satisfies the validity property. In order to prove the agreement property, we need the following two notions. A subset of nodes constitute a *path covering* of a tree if every path in the tree from the root node to a leaf node contains at least one node from the subset. A subset of nodes is *common* if all nonfaulty processes agree on the new value for every node in the subset.

Lemma 3 *After $m+1$ rounds, there is a path covering for the tree that is common.*

Proof: Consider the set C of all nodes whose label ends with the index of a nonfaulty process. Clearly, from Lemma 2, all nodes in C are common and therefore C is common. To see why C forms a path covering of the tree, note that every path from the root node to a leaf node consists of exactly $m+1$ nodes. Moreover, the way the tree is constructed, no label can contain an index (of a process) more than once. Since at most m processes are faulty, there is at least one node in the path that ends with the index of a nonfaulty process. This implies that every path from the root node to a leaf node contains at least one node from C. □

Lemma 4 *After $m+1$ rounds, if there is a common path covering for the subtree rooted at x, then x is common.*

Proof: The proof is by induction on the length of x.

Base Case $[|x| = m+1]$**:** In this case, x is a leaf node. Consequently, any path covering for the subtree rooted at x contains x. If that path covering is common, then, trivially, x is common.

Induction Step $[|x| = r$**, where** $1 \leqslant r \leqslant m]$**:** Let C be a common path covering for x. If x is in C, then, clearly, x is common. Therefore assume that x is not in C. Consider a child xk of x. Now, C induces a common path covering for xk. By induction hypothesis, xk is common. Since xk was choosen arbitrarily, all children of x are common. Therefore all nonfaulty processes agree on the new value for x. This in turn implies that x is common. □