Lecture 7

Distributed Coordination and Agreement

(07-02-2009, Chapter 12)

Lecture Outline

1. Introduction
2. References
3. Centralized synchronization algorithms
4. Distributed synchronization based on timestamps
5. Distributed synchronization based on token passing
6. Election algorithms
7. Distributed Consensus Problem

1. Introduction

(1) Problem description

a. Coordination

b. Agreement

Same in nature

(2) Variants of the problem

2. References

(a) A. Silberchatz, J. Peterson, and P. Galvin, *Operating sytem concepts*, Chapter 13.2, 13.7, (2nd ed.), Addison-Wesley, 1991.

(b) M. Raynal, *Distributed algorithms and protocols*, Chapter 2, John Wiley, 1988.

(c) G. Ricart and A.K. Agrawala, *An Optimal Algorithm for Mutual Exclusion in Computer Networks*, Comm. of ACM, 24(1), Jan. 1981, pp.9-17.

(d) E.G. Chang and R. Roberts, *An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processors*, Comm. of ACM, 22(5), May 1979, pp. 281-283.

(e) E.G.S. Hirschberg and J.B. Sinclair, *Decentralized Extrema Finding in Circular Configurations of Processors*, CACM, 23(11), Nov. 1980, pp. 627-628.

3. Centralized synchronization algorithms

- A centralized approach based on *central process*

- One of the node, called the*coordinator*, is designated to coordinate the entry to critical region.
- Each process that wants to enter its critical region sends a *request* message to the coordinator.
- Everytime the coordinator receives a request message, it checks if some other process is within the CS. If not, the coordinator informs its approval by sending a *reply* message. Otherwise the coordinator puts the request into a waiting queue.
- Once a process exits its critical region, it send a *release* message to the coordinator.
- Everytime the coordinator receives a release message, it checks if the waiting queue is empty. If not, it removes one of the waiting request from the queue and sends *reply* message to the requesting process. The above algorithm satisfies all the three requirements of CS solutions if the properly queue is properly maintained (say as a FIFO). Each entry to the CS requires three messages: *request, reply*, and *release.*

  One severe problem is what happen if the coordinator goes down. The whole system would halt. We must be able to detect the failure of the coordinator and elect a new coordinator.

4. Distributed synchronization based on timestamps

   Distributed approach – Decision-making is distributed accross the whole system.

   - When a process P_i wants to enter its CS, it generates a new timestamp, *TS*, and sends the message *request*(P_i, *TS*) to all other processes in the system (including itself).
   - On receiving a *request* message, a process may reply immediately by sending a *reply* message to P_i, or it may delay sending a reply.
   - A process that has received a *reply* message from all other processes in the system can enter its CS, queueing incoming requests and defering them. After exiting its CS, the process sends reply to all its defered requests.
   - The decision whether a process P_j replies immediately to a request *request*(P_i,*TS*) message or defers its reply is based on three factors:

     a. If P_j is in its CS, then it has to defer the reply.

     b. If P_j does not want to enter its CS, then it sends a *reply* immediately to P_i.

     c. If P_j wants to enter its CS but has not entered yet, then it compares its own request timestamp with the timestamp *TS* of the incoming request from P_i. If its own request timestamp is greater than *TS*, then it sends *reply* immediately (P_i asked first). Otherwise, the reply is defered.

- The algorithm has the following desirable behaviour:

  a. Mutual exclusion is preserved and freedom from deadlocks is ensured.

  b. Freedom from starvation is ensured, because entry to CS is scheduled according to the timestamp ordering. The timestamp ordering ensures that the requests are served in a first-come-first-served order.

  c. The number of messages per CS entry is $2 \times (n-1)$.

- **Example**. Three processes P_1, P_2, P_3. Assume P_1 and P_3 want to enter their CSs. P_1 sends a message *request*$(P_1,\ TS = 10)$ to P_2 and P_3, while P_3 sends message *request*$(P_3, TS = 4)$ to P_1 and P_2. (The timestamp values 10 are 4 are obtained from the logical clocks). When P_2 receives the two requests, it sends its replies back immediately. P_1 receives the request from P_3, it also sends its reply immediately, because its own timestamp 10 is greater than the timestamp 4 of P_3's request. When P_3 receives request from P_1, it defers its reply. On receiving replies from both P_1 and P_2, P_3 enters CS. After exiting from its CS, P_3 sends reply to P_1, and P_1 can enter its CS.

- Three problems:

  i. The processes must know the identities of all other processes in the system. When a new process joins the group of processes, the following actions need to be taken:

     a. The process must receive the identities of all other processes in the group.

     b. The name of the new process must be sent to all other processes.

     Not a trivial task.

  ii. If one of the processes fails, the entire algorithm is defeated. We can monitor the states of all processes in the system, and notify all active processes once a process failure is detected.

  iii. Processes that have not enter its CS must pause frequently to participate the decision making. Therefore this protocol is only suitable for a small, stable set of processes.

- Complexity: needs $3(n-1)$ messages in a system with n processes.

5. Distributed synchronization based on token passing

(1) General ideas:

A single token is circulating around the distributed system. Only the process that is holding the token is allowed to execute its C.S.

(2) The algorithm by Ricart and Agrawala (CACM 1979)

a. **Assumptions**:

∗ The system is fully linked.

∗ The transmission is error-free.

∗ Transmission is variable.

∗ Desequencing is possible – messages may be recieved in an order different from that in which they were sent.

b. **Principles**:

(a) Like any token-based algorithm, a single token is circulating. Token holder can enter its C.S. without asking permission from other processes.

(b) Initially the token is assigned arbitrarily to one of the n processes. A process P_i that wishes to enter its C.S. broadcasts to all other processes a request message m that contains a timestamp T_i.

(c) The token is implemented as a vector of n components. The i^{th} component of the token records the timestamp when the last time the token was assigned to process P_i.

(d) When a process, say P_j, which is holding the token, no longer needs the token, it will search the entries $j+1, j+2, \cdots, n, 1, 2, \cdots$ of the token for the first value l such the timestamp of P_l's last request for the token is greater than the value recorded in the token for the timestamp of P_l's last holding of the token. P_j then sends the token to P_l.

c. The algorithms. The following data structures are maintained in each process P_i:

```
clock:  0, 1, ..., initialized 0; (logical clock)
token_present:    boolean;
token_held:       boolean initialized F;
token:            array (1,2,..., n) of (0,1,...) initialized 0;
request:          array (1,2,..., n) of (0,1,...) initialized 0;
```

The boolean variable 'toke_present' is initialized to F in every process except one, which is holding the token at the start.
The operation **wait**(access, token) causes process to wait until a message of the type 'access' is received.

The algorithms has two parts. The first part deals with the use of C.S. and consists of a prelude section, followed by the C.S. and ending with a postlude section. The second part deals with the actions to be performed when messages are received.

```
{Prelude}
if ¬ token_present then begin clock ← clock+1;
                               broadcast(request, clock, i);
                               wait(access, token);
                               token_present ← T;
                         end;
endif;

token_held ← T;
    <critical section>

{Postlude}
token(i) ← clock;
token_held ← F;
for j from $i + 1$ to $n$, 1 to $i - 1$ do
        if request(j) > token(j) ∧ token_present then
          begin
            token_present ← F;
            send(access, token, j);
          end;
        endif;

when received(request, k, j) do
        request(j) ← max(request(j), k);
        if token_present ∧ ¬ token_held then
          <text of postlude>
        endif;
enddo
```

Notes:

- If the token is not present, execution of the prelude section will broadcast the request to all the other processes and wait until the token arrives.
- The postlude section first records in the token the time of its last holding by P_i, looks to see if any process has requested the token and if so transfers it appropriately.
- The receipt of a request from P_j has the effect of updating the local variable 'request(j)' which records the time of P_j's last request, followed by the transfer of the token if it is neither held nor being used by any other process.
- Correctness is guaranteed by showing that at any time instance, only exactly one variable token_present has value T. This can be proved by induction. Initially, it is true. The prelude for process P_i changes the variable token_present

from F to T when it receives the token. The process P_j that sent the token must have changed its variable token_present from T to F before sending it. This establishes the mutual exclusion property. (all variables token_present have value F when and only when the token is being transmitted).

- Fairness and freedom from deadlocks.
  - Freedom from deadlock can be proved by contradiction argument
  - Fairness is guaranteed by (1) all messages are delivered within a finite time of issue. The postlude requires that P_i transfer the token to the first process P_l, found in scanning the set of in the order $l = i+1, i+2, \cdots, n, 1, \cdots i-1$, whose request has reached P_i. If the transmission delays are finite (no messsge lost), all the processes will learn of the wish of some P_j to enter its C.S. and will agree on this when its turn comes.

6. Election algorithms

(1) Why do we need election algorithms and what is an election algorithms?

- Distributed algorithm with centralized controls: when the coordinator fails, the whole algorithm fails.
- Some node is providing some special service to the system. Failure of such node should be detectd.
- Normally the nodes are numbered and the node with the largest id will be the coordinator. If the coordinator fails the node with the largest id in the remaining active nodes will be elected.

(2) The Chang and Roberts Algorithm (CACM 1979)

a. Assumptions:
   - Applicable to a system connected by a unidirectional ring.
   - Each process has a unique id, and the process with the largest id is the coordinator.
   - Each process knows its own id but may not know others ids.

b. General ideas. Based on the principle called "selective extinction".
   - The election procedure can be initiated by any process P_i. First P_i marks itself and send an *election* message containing its id to its left neighbor.
   - Every time an unmarked process receives the election message, it will compare the id in the message with its own id, and sends out the larger of two. It will then mark itself.
   - When a marked process receives the election message with its own id in it, it knows that it is elected and will broadcast an *elected* message to all other processes.

c. The algorithm:

```
Each process $P_i$ maintains the following local variables and constants:
        constant   my_number:   value i;
        varaibles  participant: boolean initialized F;
                   co-ordinator: integer;

when decision (initiate_election) do
        participant ← T;
        sendL(election, my_number);
when received (election, j) do
        case j > my_number then begin
                sendL(election, j);
                participant ← T;
                end case
        case j < my_number and ¬ participant then begin
                        sendL(election, my_number);
                        participant ← T;
                end case
        case j = my_number then begin
                        sendL(elected, i);
                end case
when received (elected, j) do
                co-ordinator ← j;
                participant ← F;
                if j ≠ my_number then sendL(elected,j);
                end if
end do;
```

d. Time complexity: two extreme cases:

- ∗ Every process initiates an election at the same time. The largest number is found in one turn round the ring. Time needed is proportional to n.
- ∗ The process immediate after the largest process P_{max} initiates the election. It takes $n-1$ transmissions for P_{max} to receive the election message, another $n-1$ message for the largest number to come back to the largest process. All together, time needed is still $O(n)$.

e. Message complexity:

i. The most favorable situation is one the processes are arranged in increasing order of their processes id's and the election is initiated by the largest process P_{max}. Only $2n$ messages are needed.

ii. The worst situation is one the processes are arranged in decreasing order of their processes id's and all initiate the election at the same time. In this case the election message issued by process P_i can travel i hops. So the total number of messages

$$\Sigma_{i=1}^{n} i = \frac{1}{2} n(n+1)$$

So the message needed is $O(n^2)$.

iii. Average case. Let $P(i,k)$ be the probability that the election message issued by process P_i can travel $k-1$ hops, i.e. the $k-1$ processes after P_i has labels less than that of P_i and the k^{th} process after P_i has label larger than that of P_i. Then

$$P(i,k) = \frac{C(i-1,k-1)}{C(n-1,k-1)} \times \frac{n-i}{n-k}$$

Where $C(i,k)$ is the number of *combinations* k items from a set of i items. The mean number of transfers from process P_i, $i < n$ is

$$E_i = \Sigma_{k=1}^{n-1} k P(i,k)$$

The mean number of total messages is

$$E = n + \Sigma_{i=1}^{n+1} E_i$$

Which can be shown is $O(n \log n)$.

(3) The Hirschberg and Sinclair Algorithm (CACM 1980)

a. **Assumptions**:

∗ The processes are organized around a bidirectional ring in arbitrary order.

∗ Each process has a unique numerical identifier. Each process may not know the identifies of its two neighbors.

b. **General Ideas**:

∗ The algorithm is based on the idea of constructing a sequence of elections on increasingly larger subsets of processes until all the processes have been included.

∗ The election procedure can be initiated by any process P_i. Process P_i initiates an election by declaring itself as a candidate and sending its id to its two neighbors, say P_j and P_k. P_j and P_k will compare its own id with the P_i's id and replace P_i as new candidates if their id's are larger than P_i's. Otherwise, P_i will remain to be the candidate, and P_i will test the legitimacy over a range of 4 processes (P_k, P_j, and neighbors of the two).

∗ Every time a round of test is successful the range of tests is doubled.

- ∗ If the test if not successful, process P_i will remain *passive* from that point on and pass incoming messages.

c. The algorithms.

The following communication primitives are used in the algorithm

sendLR send the same message to both the left-hand and right-hand neighbors.

pass pass the message received from right-hand neighbor to left-hand neighbor and vice versa.

respond send a response to the neighbor from which a message has just been received.

The following data structures are maintained in each process P_i:

constant	my_number**value** i;	
variables	state:	**position initialized** not_involved;
	lgmax:	**integer**;
	winner:	**integer**;
	nbresp:	0,1,2 **initialized** 0;
	respOK:	**boolean initialized** T;

The type **position** is defined as

type position = (not_involved,candidate,lost,elected)

Three types of messages (two for elections):

- ∗ **(candidate,number,lg,lgmax)**, where:

  number is the identifier of the process sending the candidacy message.

  lg is the distance (length) around the ring already traveled by this message.

  lgmax is the maximum distance the message can travel.

- ∗ **(response,bool,number)**, associated with **pass** and **response**.

  bool is T if the response is favorable ('respOK'), F otherwise.

  number is the identifier of the destination process (therefore is a means of addressing on the ring, since id's are unique).

- ∗ **(ended, number)**, used to indicate the election ends.

  number is the id of the elected process.

```
when decision (initiate_election) do
          state $\leftarrow$ candidate;
          lgmax $\leftarrow$ 1;
          while state = candidate do
              nbresp $\leftarrow$ 0; respOk $\leftarrow$ T;
              sendLR(candidate, my_number, 0, lgmax);
              wait nbresp = 2;
              if $\neg$respOk then state $\leftarrow$ lost;
              endif;
              lgmax $\leftarrow$ 2*lgmax;
          endwhile;
end do;
when received (response, bool, number) do
      if number = my_number then
              nbresp $\leftarrow$ nbresp + 1;
              respOK $\leftarrow$ respOK$\wedge$bool;
      else pass(response, bool, number);
      end if;
end do;
when received (candidate, number, lg, lgmax) do
      case number < my_number then
              respond(response, F, number);
              if state = not_involved then initiate_election;
              end if;
          end case;
      case number > my_number then
              state $\leftarrow$ lost;
              lg $\leftarrow$ lg + 1;
              if lg < lgmax then
                  pass(candidate, number, lg, lgmax);
              else respond(response, T, number);
              end if;
          end case;
      case number = my_number then
              if state $\neq$ elected then
                  state $\leftarrow$ elected;
                  winner $\leftarrow$ my_number;
                  pass(ended, my_number);
              end if;
          end case;
end do;
```

```
when received(ended, number) do
        if winner $\neq$ number then
                pass(ended, number);
                winner ← number;
                state ← not_involved;
        end if;
end do;
```

d. Time and message complexities: two extreme cases.
Assume that n is a power of 2, i.e. $n = 2^l$ for some $l \geq 0$.

* Only P_{max} initiates the election. It will take $l+1$ rounds of testing before P_{max} receives its own testing message. During the i^{th} ($0 \leq i \leq l$) round, a message issued will travel to the process 2^i processes away from P_{max}. Therefore 4.2^i messages will be transmitted. The total number of messages transmitted is

$$4(1 + 2 + 2^2 + \cdots 2^l) = 4(2^{l+1} - 1) = 8n - 4$$

The time needed is also bounded by $8n - 4$ because message transmissions might be serial. So both time and message complexities are $O(n)$ in the best case.

* More than one process initiates election. Consider the following facts:

Fact 1: A candidacy message issued by process P_j during the i^{th} round can at most travel to a process that is 2^i processes away from P_j.

Fact 2: P_j can launch a test on a range of $22^i + 1$ processes if and only if P_j has successfully tested its candidacy on a range $22^{i-1} + 1$ processes.

Fact 3: In any set of $2^{i-1} + 1$ processes, only one can launch a candidacy testing on a path of length 2^i.

Although in the worst case all processes can launch their candidacy testing simultaneously, we have

· At most $n/2$ that are candidates with paths of length 2;

· At most $n/3$ that are candidates with paths of length 4;

· At most $n/(2^{i-1} + 1)$ that are candidates with paths of length 2^i.

The two candidacy testing messages issued by a process on a path of length 2^i will incur 42^i messages. The total number of messages is

$$4 \times (1.n + 2\frac{n}{2} + 4\frac{n}{3} + \cdots + 2^i \frac{n}{2^{i-1} + 1} + \cdots + 2^l \frac{n}{2^{l-1} + 1})$$

Each term in above summation is less than $2n$. Since we have $l+1 = 1+\log n$ terms, the number of total messages is bound above by $4.2n.(1+\log n) = 8n(1+\log n)$. So the number of messages is $O(n \log n)$. Time complexity is also $O(n \log n)$ because at most $8n(1+\log n)$ time units are needed to transmit $8n(1+\log n)$ messages.

7. Distributed Consensus Problem

(1) Introduction

(2) Failures in DS: they may cause components to behave unpredicably – sending conflicting information to different parts of the system.

(3) Synchronization vs. broadcast in DS

a. Broadcasting has been used by many DSs as a basic mechanism for sending an item of information from one process (the issuer) to a number of other processes (the receivers).

b. Synchronization problems in DS:

(a) The critical section problem (CSP)

(b) The problem of managing transactions in DSs

(c) The election problem

c. Effects of failures: they complicate the above problem.

(4) The distributed consensus problem.

a. This problem is also a synchronization problem. However it is a more general synchronization problem than the CSP, transaction management, or, election problem, because processes in this problem can be *faulty* – they can send conflicting info in response to a synchronization request.

b. Problem description: A network of n processes can communicate with each other only by means of message transmission over bidirectional comm. channels. It is assumed that reliable broadcasting of messages is available, i.e. when any process issues any item of information (i.e. message), all other processes will receive that item unchanged (i.e. have the same "perception" of the item). We say that a consensus has been reached if all the processes that are operating *reliably* reached the same conclusion.

c. Note:

(a) If the issuing process is reliable, then it will send (broadcast) the same message to every other processes. If the issuing process is faulty, however, it may send different messages to different processes, or even may not send a message to a process.

(b) A receiving process does not know if a sending process is reliable or not.

(c) This problem has also been called the *Byzantine Generals Problem*, or the *Unanimity Problem* in the literature.

(d) An analogue of this problem (the reason why it is also called the Byzantine Generals problem) is: In ancient time, a group of Byzantine generals surrended an enemy's city. The generals had to reach a consensus about the next step (attack time, withdraw time, and etc.). There were two types of generals: loyal generald and traitorous generals. Any of the generals could initiate a proposal of next step action and such a proposal would be transmitted among the generals using messengers. Messengers were reliable – they wouldn't change the message contents. A traitorous general on the other hand could change the message content before sending it to next general. The problem is the following: how could we let the loyal generals reach the same conclusion of the next step action, regardless the sabotage of the traitorous generals?

(5) Features of solutions to the problem

a. Due to the possibility that the issuing process may be faulty, and the requirements that the reliable processes must agree among themselves on the common decision, the receiving processes have to exchange values they received in order to achieve consensus.

b. We can assume that there is a function *majority* available to each reliable process, which will apply the function to messages received. This function will return same value as long as the parameters (i.e. messages) are the same. The ordering of the parameters is immaterial.

c. If the maximum number of simultaneously faulty processes is t, then any solution to the problem will need at least $t + 1$ stages of message exchanges in order to arrive at a consensus.

d. Measurements of Solutions:

(a) The number r of stages of exchanges necessary to arrive a consensus (mimimum is $t + 1$ where t is the number of faulty processes).

(b) The total number of messages that have to be exchanged.

(c) The maximal length m of messages exchanged.

(6) Impossibility result

Theorem There is no solution to the distributed consensus problem unless the maximum number t of faulty processes at any time is strictly less than one third of the total number n of processes in a network. Namely:

$$n \geq 3t + 1$$

Proof: Induction on t and n. For simility, we only consider the case $t = 1$ and $n = 3$ and show that no consensus is possible in this case.

Let P_0, P_1, and P_2 are three processes and P_0 be the issuing process.

Case 1 : P_2 is the faulty process. See Figure 32. Here P_1 receives (a, b) and its conclusion should be a (same as the other reliable process P_1).

Case 2 : P_0 is the faulty process. See Figure 33. Here P_1 receives (a, b) again and its conclusion should be a according Case 1. P_2 receives (a, b) also and concludes to a too.

Case 3 : P_1 is the faulty process. See Figure 34. Here P_2 receives (a, b) again and its conclusion should be consistent to a (Case 2). However, the other reliable process is P_0 and its conclusion is b (it issues b in the first place). Therefore a contradiction arises: P_2 has to conclude both a and b for the same messages sequence (a, b) received.

Note: Fig.12.19, p.505 shows similar argument for impossibility of three generals.

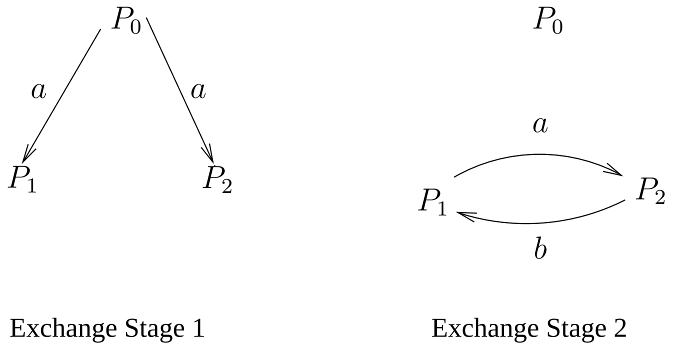


Figure 32: Case 1 of Impossibility Result Proof

(7) The Lamport, Shotstak, and Pease algorithm (ACM TOPLAS, July 1982)

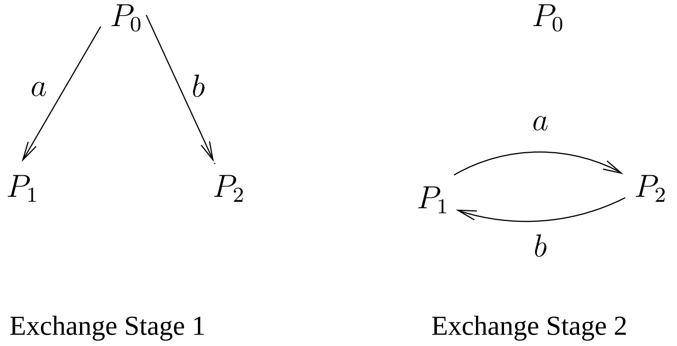


Figure 33: Case 2 of Impossibility Result Proof

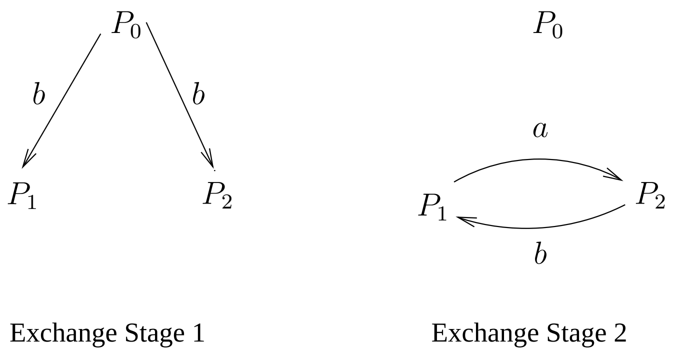


Figure 34: Case 3 of Impossibility Result Proof

a. Assumptions about the network

  (a) There are n processes, and t of them are *faulty*;

  (b) The comm. environment through which the processes exchange messages is reliable;

  (c) The comm. channels are bi-directional and the network topology is a complete graph (i.e. every process is directly linked with every other process);

  (d) Messages are transmitted in FIFO order.

b. Assumptions about the messages

  (a) A faulty process may modify the contents of an incoming message before forwarding;

  (b) The id of the issuer of a message is unknown to the receiver;

  (c) A faulty process may elect not to forward a message after receiving it. In this case, we assume there is a special message v_{def}, which a process P_i may assume is from P_j if P_j doesn't forward message to P_i in a specific stage of message exchanges.

c. Outline of the algorithm

  (a) The algorithm consists of a sequence of stages of message exchanges. This is true for all algorithms for solving this problem. Cf. the note in 1.(3).c of this lecture.

  (b) If upto t processes can be faulty, then at least $t+1$ such stages are necessary. This fact can be easily deduced from the impossibility result Theorem.

  (c) This is a recursive algorithm. For similicity of presentation and without loss of generality, we assume the issuer is always P_0.

  (d) The first stage consists of the issuer P_0 sending out the value v to every other processes. In the subsequent stages, these "other processes" exchange the values they receive. Every process attaches its id to every message it sends out (so that the receiver processes will not exchange that same message with the original sender). Each message is of the form:

$$(v, P_0, P_{i_1}, P_{i_2}, \cdots, P_{i_{n-1}})$$

Notice that inculding P_0 as part of the message is not really necessary. The rule is that, once P_i receives a message, it will forward the message to every process listed in the message, excluding itself and the original sender.

(e) All reliable processes exchange messages unchanged. Those faulty processes may alter a message before exchanging it with others, or elect to withhold the message.

(f) Assume there is a function *majority* that will return a value from its given parameters. The order of parameters for this function is not important. For example, *majoriry*(a, b, c) is equal to *majority* (c, a, b). At the end of message exchange stages, each process will invoke this function with message values received from other processes as parameters. The value returned will be the conclusion reached by that process.

d. Algorithm $UM(n, t)$

```
begin
  (a) Issuer process $P_0$ sends its value $v$ to each other $n-1$ processes $P_1, P_2, \cdots, P_{n-1}$.
  (b) Each receiver process notes the value received from $P_0$, or records the default
      value $v_{def}$ if it has not received anything.
  (c) if $t > 0$ then
      begin for every process $P_i$ do
        i. Let $v'$ be the value noted by $P_i$ in Step b. $P_i$ acts as the issuer, and sends $v_i$
           by calling $UM(n-1, t-1)$ to all other $n-2$ processes $P_1, \cdots, P_{i-1}, P_{i+1}, \cdots, P_{n-1}$.
           The message sent is of the form:

               $(v', P_i, P_1, \cdots, P_{i-1}, P_{i+1}, \cdots, P_{n-1})$

        ii. Let $v_j$ $(i \neq j)$ be the value received by $P_i$ from $P_j$ at the end of Step
            b.(ii), or $v_{def}$ if nothing is received. $P_i$ records its conclusion as value

               $majority(v_0, v_1, \cdots, v_{i-1}, v_{i+1}, \cdots, v_{n-1})$

        end
      endif
end
```

e. Correctness of the algorithm: can be proved by induction on n and t and is omitted.

f. Example. Consider the case, $n = 4$, $t = 1$. We invoke the function $UM(4, 1)$.

(a) Case 1. P_3 is faulty. Figure 35 shows the messages exchanged. At the end of Stage 1:

  - At P_1, $v_0 = a$;
  - At P_2, $v_0 = a$;
  - At P_3, $v_0 = a$;

  At the end of Stage 2:

  - At P_1, $v_0 = a$, $v_2 = a$, $v_3 = b$;
  - At P_2, $v_0 = a$, $v_1 = a$, $v_3 = c$;
  - At P_3, $v_0 = a$, $v_1 = a$, $v_3 = a$.

  At the end of the algorithm both P_1 and P_2 have the same conclusion (*majority*(a, a, b) and *majority*(a, a, c) respectively), which is also is P_0's conclusion a. Notice that the conclusion of P_3 is not important, because it is a faulty process anyway.

(b) Case 2. P_0 is faulty. It issues a to P_1, b to P_2, and does not send anything to P_3. Figure 36 shows the messages exchanged. At the end of algorithm, all three reliable processes P_1, P_2, and P_3 have the same conclusion *majority*(a, b, v_{def}).

Note: Fig.12.20, p.507 shows similar argument for $n = 4$ and $t = 1$ where the faulty process is p_3.

g. Complexity of the Algorithm

(a) Number of stages: $t+1$. This is optimal in light of of the impossibility result. It can tolerate t faulty processes when $n > 3t$ holds.

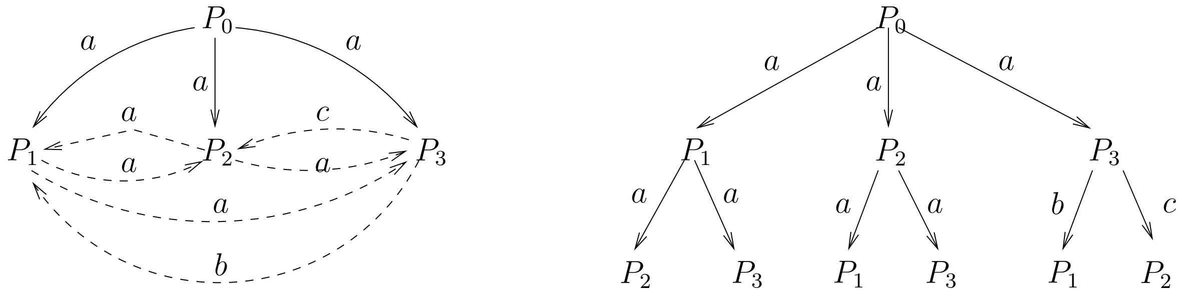
(b) Number messages

  - Stage 1: $UM(n, t)$ will generate $n - 1$ messages;
  - Stage 2: there are $n-1$ $UM(n-1, t-1)$ function calls, each will generate $n - 2$ messages, for a total $(n - 1)(n - 2)$ messages;
  - Stage 3: there are $(n - 1)(n - 2)$ $UM(n - 2, t - 2)$ function calls, each will generate $n - 3$ messages, for a total $(n - 1)(n - 2)(n - 3)$ messages;
  - ...
  - Stage t will generate a total $(n-1)(n-2)\cdots(n-t)$ and Stage $t+1$ (the last stage) will generate

  $$(n - 1)(n - 2) \cdots (n - t)(n - t - 1)$$

  messages.

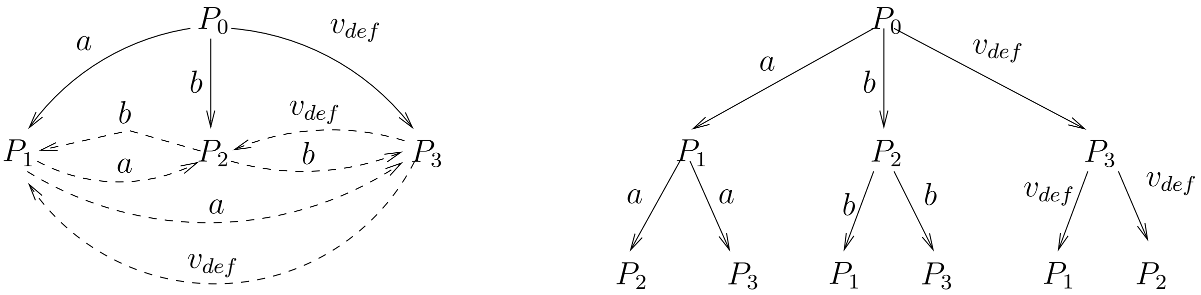
  - Therefore the total number of messages is $(n - 1) + (n - 1)(n - 2) + (n - 1)(n - 2)(n - 3) + \cdots + [(n - 1)(n - 2) \cdots (n - t)(n - t - 1)] = O(n^{t+1})$.



(a) Message Exchanges
P_3 Faulty

(b) Another View of Message Exchanges

Figure 35: Case 1 of Example: P_3 Faulty



(a) Message Exchanges
P_0 Faulty

(b) Another View of Message Exchanges

Figure 36: Case 2 of Example: P_0 Faulty

(c) Message sizes: at stage k, each message contains a sequence of k process ids, therefore is proportional to k.

(8) Other algorithms

	# of Faulty Processes	# of Stages	Maximal Message Size
Optimal Case	$n \geq 3t + 1$	$t + 1$	1
LSP Algorithm	$n \geq 3t + 1$	$t + 1$	t
Berman & etl. Algo.(1989)	$n \geq 3t + 1$	$t + 1$	polynomial
Berman & etl. Algo.(1989)	$n \geq 3t + 1$	$3t + 3$	2
Moses & etl. Algo.(1988)	$n \geq 6t + 1$	$t + 1$	polynomial