University of Freiburg, Germany
Department of Computer Science

Distributed Systems

Chapter 4 Coordination and Agreement

Christian Schindelhauer

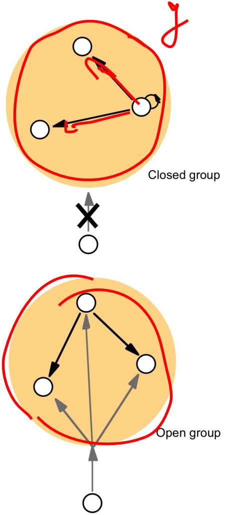
19. May 2014

4.4: Multicast communication

- With a single call of *multicast*(g, m) a process sends a message to all members of the group g
- Using *deliver*(m), received messages are delivered on participating processes
- *Efficiency*
  - Number of messages, transmission time
- *Delivery guarantees*
  - ordering
  - receipt
  - e.g. IP Multicast does not guarantee ordering of success

4.4: Multicast communication

- *System Model*
  - *multicast*(g, m): sends the message m to all members of group g
  - *deliver*(m): delivers a message to the process (message has been received by lower level)
  - *sender*(m): sender of a message m (within the message header)
  - *group*(m): group of a message m (within the message header)
- Allowed senders
  - closed group: senders must be members of a group
  - open group: any process can send a message to the group



Basic Multicast

- *B-multicast*(g, m): for each process $p \in g$, *send*(p, m)
- *B-deliver*(m): if message m is received at p return the message m

Ack Implosion

- if too many processes participate
- if *send* uses acknowledgments, some of them could be dropped
- then the messages could be retransmitted
- further *acks* are lost due to full buffers etc.

Reliable Multicast

- *Safety: Integrity*
  - Every message is delivered at most once
  - Receiver of m is a member of $group(m)$
  - Sender has initiated a $multicast(g, m)$
- *Liveness: Validity*
  - If a correct process multicasts a messages then it eventually delivers m (to itself)
- *Agreement*
  - If a correct process delivers m then all other processes eventually deliver m

Implementing Reliable Multicast over Basic Multicast

On initialization
  Received := {};

For process p to R-multicast message m to group g
  B-multicast(g, m);   // $p \in g$ is included as a destination

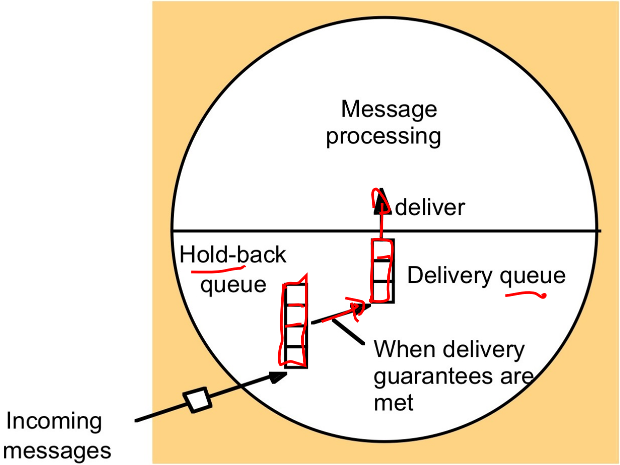
On B-deliver(m) at process q with g = group(m)
  if ($m \notin$ *Received*)
  then
    Received := *Received* $\cup$ {*m*};
    if ($q \neq p$) *then B-multicast(g, m); end if*
    R-deliver m;
  end if

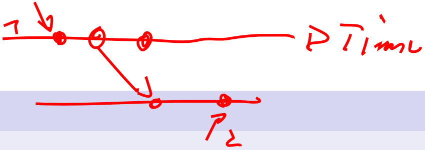
Each message needs to be sent $|g|$ times!

Implementing Reliable Multicast over IP Multicast

- *R-multicast*(g, m) for sending process p
  - Sender increments a (sending) sequence number S_g^p for group g after each messages
  - Sequence number sent with message
  - Acknowledgements of all received messages with $\langle q, R_g^q \rangle$ are piggybacked with message
  - Negative Acknowledgments: by received sequence number R_g^q causes retransmission of message
- *R-deliver*(g) for receiving process q
  - R_g^q is the sequence number of the latest message it has delivered.
  - it is sent with each acknowledgment and allows the sender (and all receivers) to learn about missing messages
  - Process q *delivers* a message m (with piggybacked S) only if $S = R_g^q + 1$.
  - messages with $S > R_g^q + 1$ are kept in a hold-back queue
  - messages with $S < R_g^q + 1$ are erased
  - After delivery $R_g^q := R_g^q + 1$

Hold-Back Queue for Arriving Multicast Messages



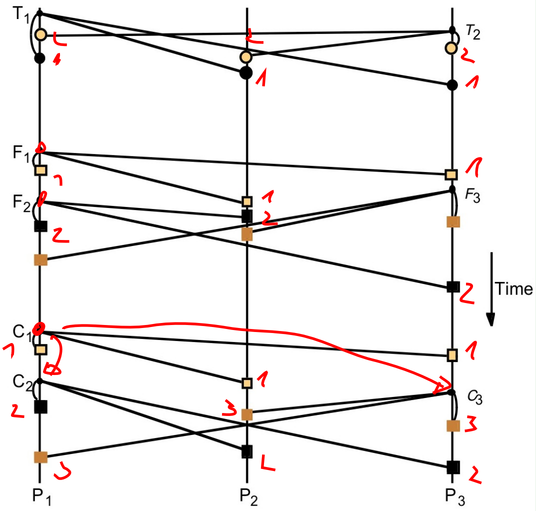


Ordered Multicast

- *FIFO Ordering*
  - If a process casts `multicast`(g, m) before `multicast`(g, m')
  - then m is delivered before m'
  - in each process of group g
- *Causal Ordering:*
  - If `multicast`$(g, m) \rightarrow$ `multicast`(g, m')
  - then m is delivered before m'
  - $\rightarrow$ is based only on messages within the group g
- *Total Ordering:*
  - If a process delivers m before m'
  - then m is delivered before m' on any other process of g

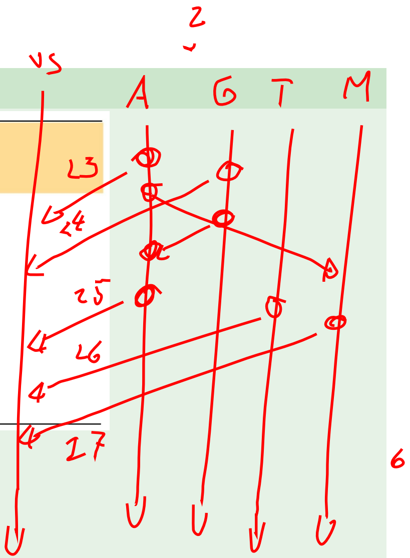
Total, FIFO and Causal Ordering

- *Total Ordering* ?
- *FIFO Ordering*
- *Causal Ordering*

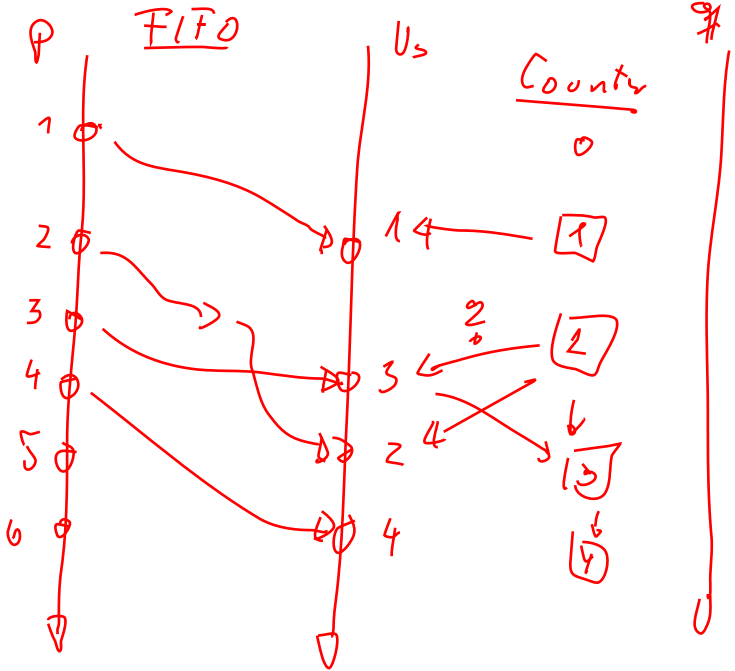


Bulletin Board

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		



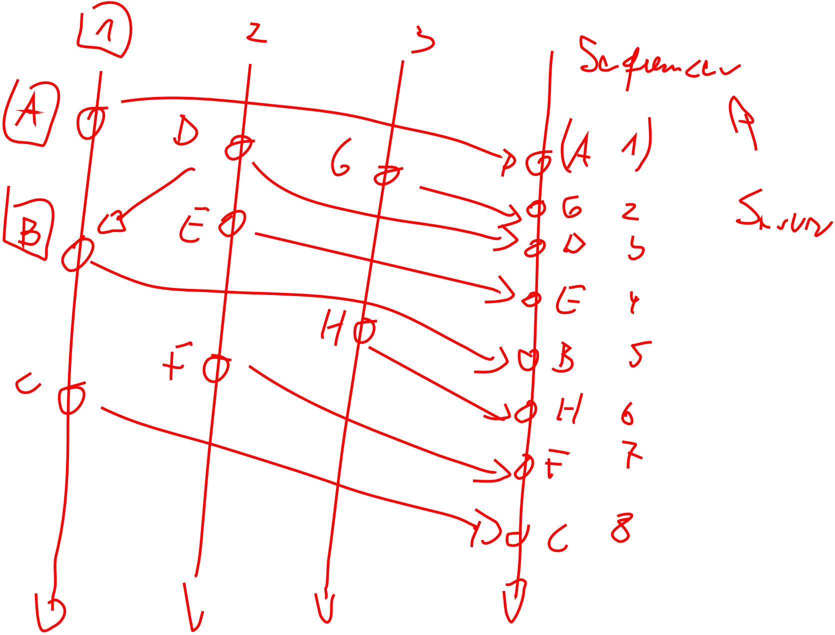
- *FIFO Ordering*
- *Causal Ordering*
- *Total Ordering*



P
FIFO
Us
Counter
0
1
2
3
4
5
6

Implementing FIFO Ordering Multicast

- Use sequence numbers for each message
  - S_g^p for each sender process p and group g
  - R_g^p for the last message delivered to process p of group g
- Multicast over IP Multicast satisfies FIFO ordering
- Essential components for FIFO ordering:
  - Sender piggybacks S_g^p on the message
  - Receiver checks wether received message satisfies $S = R_g^q + 1$
  - and delivers m and sets $R_g^q := R_g^q + 1$.
  - if $S > R_g^q + 1$ it puts m into the hold-back queue
- In combination with a reliable multicast we obtain a reliable FIFO ordering multicast algorithm


1
2
3
Sequencer
A
B
C
D
E
F
G
H
A 1
G 2
D 3
E 4
B 5
H 6
F 7
C 8

Implementing Total Ordering Multicast with a Sequencer

- A sequencer is an extra process taking care about ordering
- A sender process sends message with unique ID *i* to sequencer
- Sequencer marks message with ordering and multicasts the message

1. Algorithm for group member *p*

On initialization: $r_g := 0$;

To TO-multicast message m to group g
  B-multicast($g \cup \{sequencer(g)\}$, <*m, i*>);

On B-deliver(<*m, i*>) *with* $g = group(m)$
  Place <*m, i*> in hold-back queue;

On B-deliver(m_{order} = <“order”, *i, S*>) *with* $g = group(m_{order})$
  wait until <*m, i*> in hold-back queue and $S = r_g$;
  TO-deliver m;   // (after deleting it from the hold-back queue)
  $r_g = S + 1$;

2. Algorithm for sequencer of *g*

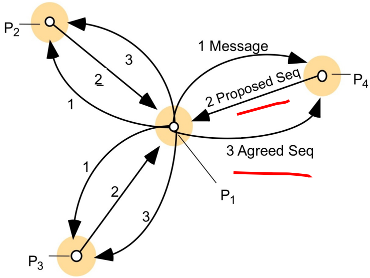
On initialization: $s_g := 0$;

On B-deliver(<*m, i*>) *with* $g = group(m)$
  B-multicast(*g*, <“order”, *i*, s_g>);
  $s_g := s_g + 1$;

no FIFO

Implementing Total Ordering Multicast using ISIS

- Used in the ISIS toolkit of Birman & Joseph
- Each participating process proposes a sequence number for a messages
  - All proposed message numbers are unique
  - The sender chooses the maximum of all proposals and sends this information (piggybacked with the next messages)
  - This agreed sequence number defines the ordering of the hold-back-queue
  - The smallest elements of the hold-back queue can be delivered as the first element
- Does not imply causal nor FIFO ordering

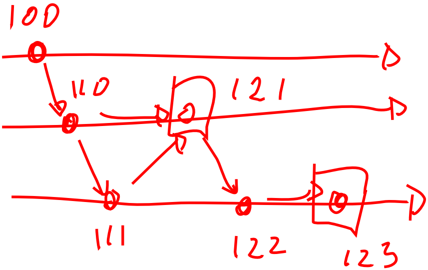


Causal Ordering

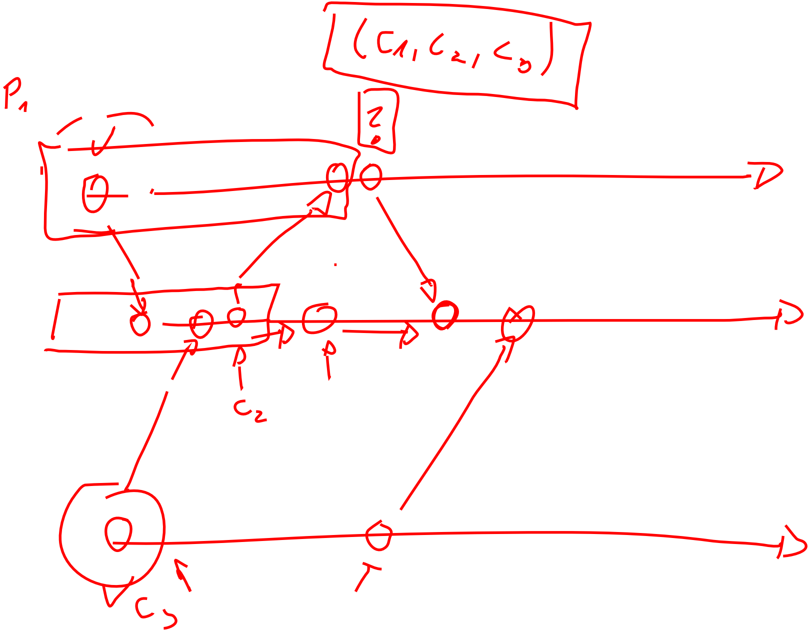
event: A B

$\boxed{A \to B \text{ ?}}$ $\Rightarrow$ **Vector Clocks**

$(1,2,3) < (1,4,5)$



$121 < 123$


(C1, C2, C3)
P1
2
C2
C3

Implementing Causal Ordering

- Uses vector clocks to keep causal ordering (piggybacked to messages)
- Vector clock $V_i^g[i]$ counts all multicast messages of process i in group g
- hold-back queue reflects vector clocks

Algorithm for group member p_i ($i = 1, 2..., N$)

On initialization

$V_i^g[j] := 0$ ($j = 1, 2..., N$);

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast(g, $<V_i^g, m>$);

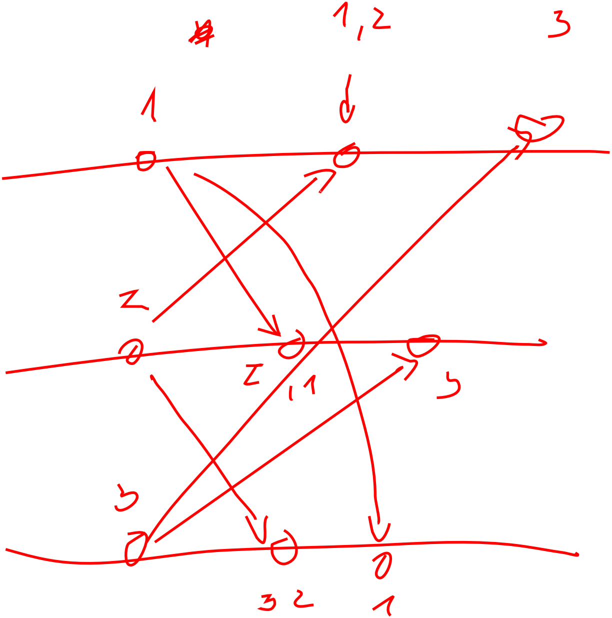
On B-deliver($<V_j^g, m>$) *from* p_j, *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ ($k \neq j$);

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;

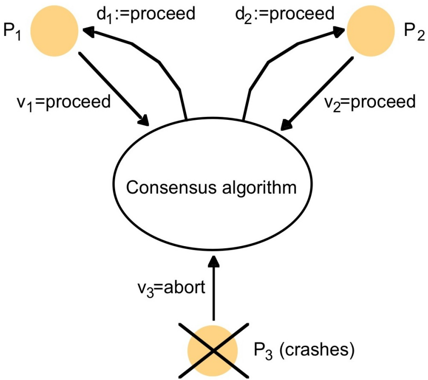


1, 2
3
1
2
2
1 1
3
3
3 2
1

4.5: Consensus

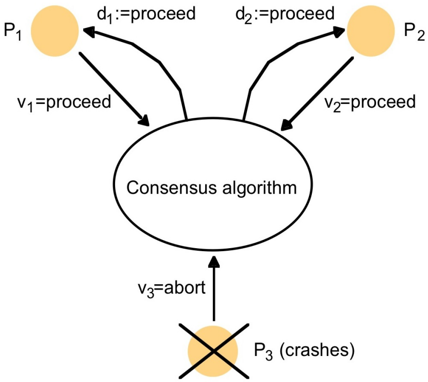
Byzantine Generals

- n processes $p_1, \ldots, p_n$
- at most f processes have arbitrary (Byzantine) failures
- Every process starts in the *undecided* state and *proposes* a value v_i
- Eventually all correct processes p_i
  - choose the *decided* state
  - and choose the same value $d_i \in \{v_1, \ldots, v_n\}$
  - and stay in this state



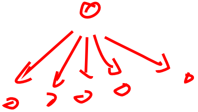
Consensus Problem

- *Termination:* Eventually each correct process p_i is *decided* by setting variable d_i
- *Agreement:* The decision value d_i of all correct processes is the same
- *Integrity:* If all correct process proposed the same value v, then $d_i = v$ for all correct p_i



- Possible decision functions: *majority, minimum, maximum*, ...
- Byzantine failures can cause irritating and adversarial messages
- System crashes may not be detected

Byzantine Generals Problem



- n generals have to agree on attack or retreat
- one of them is the commander and issues the order
- at most f generals are traitors (possibly also the commander) and have adversarial behavior
- all correct generals have eventually to agree on the commander decision if he acts correctly

~~Byz~~ BG

~~Consensus~~ Problem

- *Termination:* Eventually each correct process p_i is *decided* by setting variable d_i
- *Agreement:* The decision value d_i of all correct processes is the same
- *Integrity:* If the commander is correct then all correct processes choose the commander's proposal

Interactive Consistency

A B C D E F
(F S F F S F)
(F S F ~~X~~ ~~F~~ S)
F S

- n processes need to agree on a *vector* of values
- Each process proposes a value v_i
- A correct processes eventually decide on a vector $\boxed{d_i} = \{d_{i,1}, \ldots, d_{i,n}\}$ where

$$\boxed{d_{i,j} = v_j \quad \text{if } p_j \text{ is correct}}$$

Interactive Consistency

- *Termination:* Eventually each correct process p_i is *decided* by setting variable d_i
- *Agreement:* The decision value $\boxed{d_i}$ of all correct processes is the same
- *Integrity:* If the p_j is correct then all correct processes p_i set $d_{i,j} = v_j$

The Relationship between Consensus Problems

Assume solutions to Consensus (C), Byzantine generals (BG), interactive consistency (IC)

$$
\begin{aligned}
C_i(v_1, \ldots, v_n) &= \text{consensus decision value of } p_i \text{ for proposals } v_i \\
BG_i(j, v) &= \text{BG decision value of } p_i \text{ for commander } p_j \text{ proposal } v_j \\
IC_i(v_1, \ldots, v_n)[j] &= j\text{-th position of interactive consistency} \\
&\quad\ \text{decision vector of } p_i \text{ for proposals } v_i
\end{aligned}
$$

Solving *IC* from *BG*

- In parallel n Byzantine generals problems are solved
- each process p_j acts as commander once

$$IC_i(v_1, \ldots, v_n)[j] = \boxed{BG_i(j, v)}$$

The Relationship between Consensus Problems

Solving C from $\boxed{IC}$

- *majority* returns the most often parameter or $\perp$ if no such value exists
- for all $i = 1, \ldots, n$

min

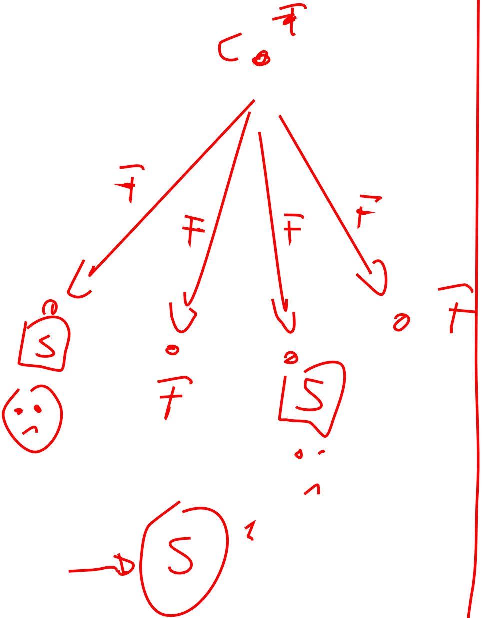
$$C_i(v_1, \ldots, v_n) = majority(IC_i(v_1, \ldots, v_n)[1], \ldots, IC_i(v_1, \ldots, v_n)[n])$$

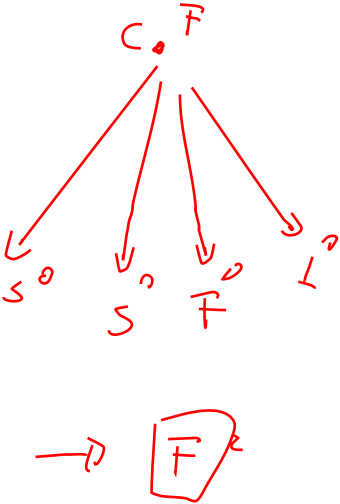
max

Solving BG from C

- The commander p_j sends its proposed value to itself and each other process
- All processes run consensus with the values $v_1, \ldots, v_n$ received from the commander
- for all $i = 1, \ldots, n$

$$BG_i(j, v) = C_i(v_1, \ldots, v_n)$$


C
F
S
F
S
S


C
F
S
S
F
L
F

$(v_1, v_2, v_3 \ldots)$

$$\underbrace{(1, 1, 2)}_{1}$$

$$\underbrace{(1, 1, 2)}_{1}$$

$$(1, \underbrace{1, 2)}_{1}$$

$P1 : 1$

$P2 : 1$

$P3 : \lightning$

Consensus in a Synchronous System

- Assume that there are no arbitrary (Byzantine) errors
- Given a synchronous distributed systems (fail-stop model)
- Use basic multicast for $f+1$ rounds
- Multicast all known values of all participants
- $Values_i^r$ denotes the set of proposed variables at the beginning of round r
- Reduce communication overhead by multicasting only freshly arrived variables $Values_i^r - Values_i^{r-1}$
- Choose the minimum of all known values as final value

Consensus in a Synchronous System

Algorithm for process $p_i \in g$; algorithm proceeds in $f+1$ rounds

On initialization

$Values_i^1 := \{v_i\};\ Values_i^0 = \{\};$

In round r $(1 \le r \le f+1)$

B-multicast$(g,\ Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round r)

{

On B-deliver(V_j) *from some* p_j

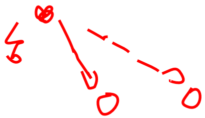
$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f+1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

Consensus in a Synchronous System



- There are no arbitrary errors only processes that crash and are correctly detected
- Given a synchronous distributed systems (fail-stop model)
- Correctness
  - Assume that two processes p_i and p_j have different values at round r
  - Then, in round $r-1$ at least one process p_k has sent different values to p_i and p_j
  - Then, p_k has crashed in this round
  - Since the number of crashes is limited to f there are not enough crashes to cover each of the $f+1$ rounds

Byzantine Generals Problem in a Synchronous System

- Assume that there are **Byzantine** errors
- Given a synchronous distributed system
  - crashes are detected
  - other wrong behavior cannot be detected, e.g. strange messages
- messages are not (digitally) signed
- at most f faulty processes

10

30

Impossibility of a solution of the Byzantine generals problem [Lamport, Shostak, Pease 1982]

- The byzantine generals problem cannot be solved for $n = 3$ and $f = 1$.
- The byzantine generals problem cannot be solved for $n \leq 3f$.

End of Section 4