Telecooperation Lab
Prof. Dr. Max Mühlhäuser

TK1: Distributed Systems - Programming & Algorithms

Chapter 3: Distributed Algorithms

Section 4: Cooperation – Multicast & Consensus

Lecturer: **Prof. Dr. Max Mühlhäuser**

Multicast Communication

Meaning? (distinguish from unicast, broadcast)

- *usually* considered: same msg sent to „several“ receivers
  - „all“ receivers (?): broadcast („whoever wants it can get it“)
  - implies receiver group; membership considered transparent to sender
  - usually, all receivers in group intended to receive msg
- essential properties: „more than syntactic sugar‘ for N * send()“
  - 1 multicast operation ≠ multiple sends
    - higher efficiency and/or
    - stronger delivery guarantees
- operations: g = group, m = message
  - *X*-multicast(g, m)
  - *X*-deliver(m)
    ≠ receive(m) (e.g., delivery only after addt‘l ctrl. msgs)
  - *X*: → additional property
    - Basic-, Reliable-, FIFO-,…

Multicast (cont.)

Major issues (cf. properties):

- addressing, in particular: „optimal routing for minimal # of msgs“
- coordination
  - guarantees that msgs are received by a group of recipients
  - delivery ordering amongst group members
- advanced: grp overlaps, membership (grp change during op.?), admission, ..

Uses of multicast, e.g.,

- Computer Supported Cooperative Work (CSCW)
  - shared WhiteBoards & applications, video-conferencing, ...
- communication w/ replicated servers (to achieve fault-tolerance)
- boosting importance: event notification, publish/subscribe systems
- discovery services in spontaneous networking

Open vs. closed: multicast group is

- closed if multicast only within (+ grp overlap forbidden → becomes broadcast algo.)
- open, if proc‘s not member of the group may send
  (work-around: first unicast to grp-member, then closed-multicast)

Multicast in IP

- only implemented by some IP routers (though part of IPv4 !): cf „mbone“
- available for UDP, uses IP auxiliary IGMP (Internet grp. mgmt.) protocol
- addressing: multicast address and port number
- IP multicast group (membership is dynamic):
  - class D IP address for which first 4 bits are 1110 in IPv4
  - IP node $\in$ multicast grp if ≥ 1 proc‘s have sockets belonging to a multicast grp
- implementation of multicast IP routers
  - efficiency 1: on LAN, use broadcast capabilities (e.g., Ethernet)
    - use locally valid multicast address, set Time To Live (TTL) counter in IP header to 1 so that packet will never get routed outside LAN
  - efficiency 2: in Internet, router forwards msgs to all other routers that have members in multicast group (late packet replication)
    - end-2-end replication: more traffic, a packet delays following ones
    - note: session directory (sd) = public domain tool (w/ GUI) allowing users to advertise multicast sessions as well as their valid multicast addresses
  - no guarantees whatsoever (msg loss, reordering, duplication, etc.)

Basic Multicast

- B-multicast = multicast with “delivery guarantee if multicasting process does not crash”
- straightforward algorithm (requires *reliable send*)

```
To B-multicast(g, m):
    $\forall p \in g$: send(p, m)
On receive(m) at p: B-deliver(m)
```

- problem in using concurrent `send(p,m)` operations: „ACK implosion“:
  - all receipients acknowledge receipt at about same time
  - buffer overflow leads to dropping of ack messages
  - retransmits, even more ack messages
- $\exists$ practical algorithm using IP-multicast

Reliable Multicast

Desired Properties:

- integrity: a correct proc p delivers a msg at most once, and the delivered msg is identical to the msg sent in the multicast send operation (safety)
- validity: if a correct proc multicasts msg m,
  then it will eventually deliver it (liveness)
- agreement: if a correct proc delivers a msg m sent to group g,
  then all other correct proc's in g will also deliver msg m
- (additionally) uniform agreement: as above,
  no matter whether initially delivering proc is correct or fails

Notes:

- validity expressed in terms of self-delivery, for simplicity reasons
  - validity and agreement amount to overall liveness requirement: if one proc (the sender) delivers a msg m, then m will eventually be delivered to all the group’s correct members
  - assumes closed multicast (no strong restriction, see above)
- agreement is similar to “atomicity”: all-or-nothing semantics
- reliable unicast supposed to have integrity & validity, too

Reliable Multicast (cont.)

Possible implementation: Realize *R-multicast* as a layer on top of *B-multicast*

- R-multicast: just B-multicast to all *proc*s in group g
- On B-deliver (@R-multicast-layer): perform a complete B-multicast again to g; *then* R-Deliver

- properties
  - validity: a correct proc will eventually B-deliver to itself
  - integrity: based on underlying communication medium
  - agreement: B-multicast to all other proc's after B-deliver
- inefficient, since each msg is sent |g| times to each proc

```
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
```

Reliable Multicast (cont.)

Reliable Multicast over IP Multicast (R-IP-Multicast):

- counters “ACK implosion” plus: may benefit from efficiency of *IP-Multicast*
- based on observation: multicast is successful *in most cases*
  - therefore: use negative acknowledgement NACK to indicate non-delivery
- Sketch of algorithm (note: assumes closed multicast groups)
  - data structures: at proc. p:
    - S_g^p: msg sequence no. for group g (# of msg‘s sent to g)
    - $\{R_g^{q_i}\}$: seq. no. of latest msg that p delivered from proc $q \in g$ (that was sent to group g)
  - init: $S_g^p := 0$; $R_g^{q_i} := -1 \quad \forall q_i \neq p$
  - piggybacking idea: when p R-IP-multicasts msg m to group g
    - it piggybacks onto m
      - S_g^p
      - $\langle q_i, R_g^{q_i} \rangle$ as acknowledgements for all $q_i \neq p$
    - *IP-Multicast* msg and piggyback information as above
    - increment S_g^p by one

Reliable Multicast (cont.)

For process p to R-IP-multicast msg m to group g:

```
    $S_g^p$ ++;
    IP-multicast (g,  <m, $S_g^p$ , <$q_1$, $R_g^{q1}$ >, ..., <$q_n$, $R_g^{qn}$ > >);    ($\forall q_i \neq p$)
                                            /* below: S,q,R without index for simplicity */
```

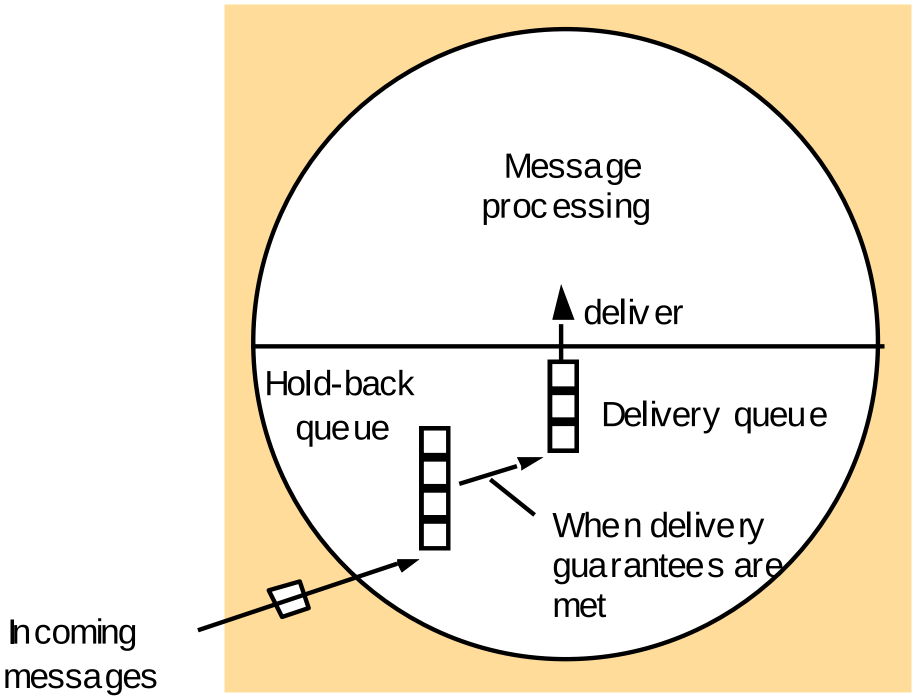
On IP-deliver (<m, S, <.,.>, ..., <q, R>, ..., <.,.> >) at process r, from process p:

```
  if    $S = R_g^p + 1$                                            /* 'expected' case */
        then    R-IP-deliver (m);
                $R_g^p$ ++;
                « check hold-back-Q »                              /* was msg a 'late' one? */
        else    if  $S > R_g^p+1$   then <store m in hold-back-Q>  /* missing msg */
                                    else    discard (m)    endif;  /* received already */
  endif;
  if    $S > R_g^p+1$  or  « $R > R_g^q$ for any process $q \in g$ »  /* $\geq$ 1 msg missing */
        then    « request missing msg's via nack's »  endif;
```

Reliable Multicast (cont.)

R-IP-Multicast (cont.):

- hold-back Q's are rather common in algo's w/ delivery guarantees
- note: piggybacked $R_g^q{}_i$ counters speed up detection of lost msg's
- note: nack may be send to neighbour close by,
  not necessarily to originator (neighbour retains copy? how long?)



Reliable Multicast (cont.)

R-IP-Multicast (cont.):

- Properties:
  - integrity
    - follows from detection of duplicates and properties of IP multicast (e.g., checksum to detect message corruption)
  - validity
    - msg loss can only be detected when a successor msg is eventually transmitted
    - requires processes to multicast msgs indefinitely
  - agreement
    - requires unbounded history for broadcast messages so that retransmit is always possible
- there exist practical variants that ensure validity and agreement

Ordered Multicast

three “subclasses”:

FIFO

if a correct process P:
    multicast(g, m);
    multicast(g, m’);
then for all correct processes:
    deliver(m’) $\Rightarrow$
        deliver(m) before deliver(m’)

Causal

if multicast(g, m) $\rightarrow_g$ multicast(g, m’)
then for all correct processes:
    deliver(m’) $\Rightarrow$
        deliver(m) before deliver(m’)

Total

if $\exists$p: deliver(m) $\rightarrow$ deliver(m’)
then for all correct processes:
    deliver(m’) $\Rightarrow$
        deliver(m) before deliver(m’)

Ordered Multicast

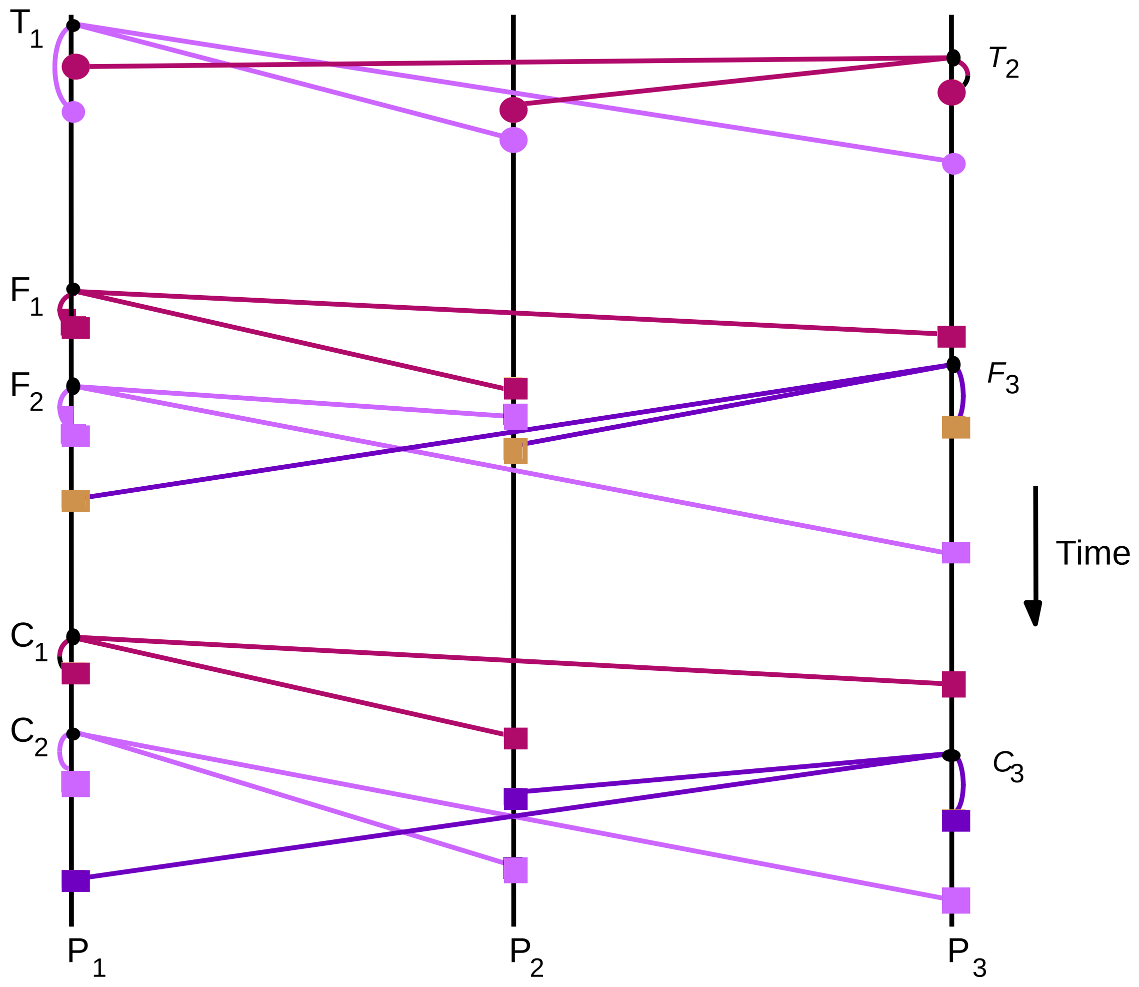
Notes:

- assume below: every process belongs to at most one group
- happen-before $\rightarrow_g$ means “restricted to communication within g”
- causal ordering implies FIFO ordering
- FIFO ordering and causal ordering are partial orders
- total order allows arbitrary ordering of deliver events relative to multicast events, as long as order is identical in all correct proc‘s
- atomic multicast: reliable, totally ordered multicast

Combinations:

- FIFO-Total = FIFO + Total
- Causal-Total = Causal + Total
- Atomic = reliable + Total

Ordered Multicast (cont.)



Notice consistent ordering of:

- totally ordered msgs T_1 & T_2
- FIFO-related msgs F_1 & F_2
- causally related msgs C_1 & C_3

- and the otherwise arbitrary delivery ordering of msgs.

Ordered Multicast (cont.)

- practical example: bulletin board:
  - FIFO makes „Hanlon‘s second posting“ a consistent statement
  - causal assures „Re: Microkernels“ appears after „Microkernels“
  - total makes „posting #24“ a consistent statement
- note: Usenet ist neither causal nor total (reason: „cost“)

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 Multicast (cont.)

FO-multicast on top of any B-multicast:

```
Data structures at process p:
    $S_g^p$ : seq.no;        $R_g^q$ : seq.no. of latest msg it has delivered from q
On initialization:
    $S_g^p = 0$
For process p to FO-multicast message m to group g
    B-multicast ( g, <m, $S_g^p$ >)
    $S_g^p$ ++
On B-deliver (<m, S >) at q from p
    if $S = R_g^p + 1$
    then    FO-deliver (m)
     $R_g^p$ ++
check hold-back queue
    else   if  $S > R_g^p + 1$ then store m in hold-back queue endif
    endif
```

Note: IP-R-multicast is FIFO & reliable!

Total Multicast

TO-multicast: (note: still assuming non-overlapping groups)

- idea: assign totally ordered IDs to multicast msgs so that every proc makes same delivery decision based on these IDs
- delivery similar to FIFO delivery, only that group-specific seq.no‘s rather than proc-specific seq.no‘s are used
- two main methods for assignment of IDs
  - sequencer
  - collective agreement on the assignment of msg IDs

A) Sequencer

- proc wishing to TO-broadcast attaches a unique ID *i* = id(m) to msg
- msg is sent to sequencer as well as all members of g
- sequencer maintains group-specific seq.no. s_g which it uses to assign increasing and consecutive seq.no‘s to the messages it B-delivers
- announces the order in which members of g have to deliver these msgs using a B-multicasted order-msg

note: sequencer is bottleneck !!

Total Multicast (cont.)

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(<"order", i, S>) with g = group(m)
    wait until <m, i> in hold-back queue and $S = r_g + 1$;
    TO-deliver m;      // (after deleting it from the hold-back queue)
    $r_g = S$;
```

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, $s_g$>);
    $s_g := s_g + 1$;
```

Total Multicast (cont.)

B) collective agreement on the assignment of msg IDs

- background: implemented in the ISIS toolkit, groups may be open or closed
- idea: receiving proc's bounce proposed seq.no.'s to sender, sender returns agreed seq.no.'s
- data structures: each proc q in group g maintains
  - A_g^q: largest agreed seq.no. it has observed so far for group g
  - P_g^q: own largest proposed seq.no.
- p B-multicasts <m, i> to g, where i is unique ID(m)
- each recipient q:
  - replies to sender with **proposal for agreed** seq.no. $P_g^q := \max(A_g^q, P_g^q) + 1$
  - as above, might yield identical seq.no for multicasts of different proc's
    $\rightarrow$ possible solution: concatenate with unique proc-ID **(see example two slides below)**
  - provisionally assigns own proposed seq.no. to msg and queues msg in hold back queue, ordered according to proposed seq.no.
- p, having received all replies:
  - chooses largest proposed number as **agreed** seq.no. *a*
  - B-multicasts <*i*, *a*> to g

Total Multicast (cont.)

B) collective agreement on the assignment of msg IDs (cont.)

- each proc q in g, having received <*i, a*> from p, sets $A_g^q := \max(A_g^q, a)$
  - reorders received msg in hold-back Q if received seq.no. differs from proposed no.
  - only when msg at head of hold-back queue is assigned an agreed seq.no., it will be queued in delivery queue

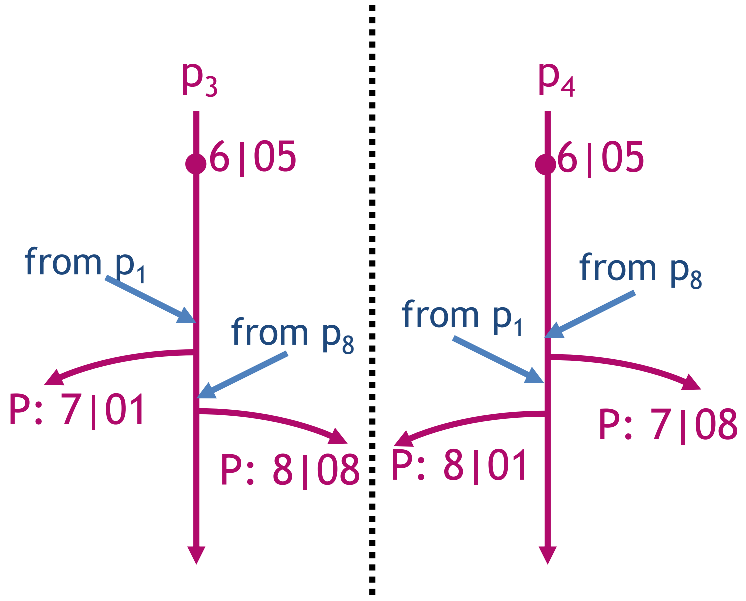
re. correctness of B (obvious: correct proc's eventually agree on seq.no.)

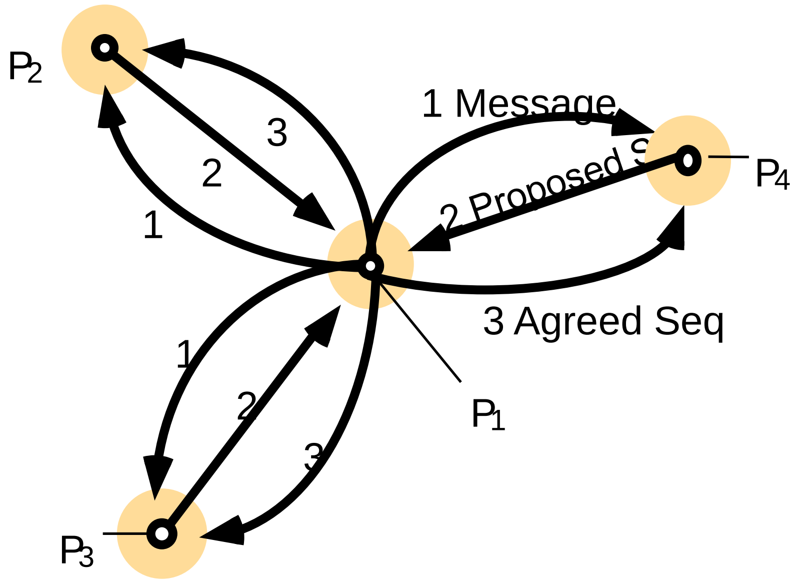
- to be shown, that
  - seq.no.s are monotonically increasing
  - no proc delivers msg before there is agreement
- assume m_1 i) to have „agreed seq.no.“; ii) to be at head of hold-back Q
  - a msg received after this state should be ‘to be delivered’ after m_1, i.e. it should be guaranteed to get a larger agreed seq.no. than m_1
  - let m_2 msg, not yet w/ agreed seq.no., but in same Q as m_1
  - $\text{agreedSeq}(m_2) \geq \text{proposedSeq}(m_2)$ (by construction of algorithm)
  - $\text{proposedSeq}(m_2) > \text{agreedSeq}(m_1)$ (since m_1 at head of queue)
  - $\Rightarrow \text{agreedSeq}(m_2) > \text{agreedSeq}(m_1)$

Total Multicast (cont.)

Example to the right:

- concatenate seq. no. (1,2,3...) with proc-ID of sender, e.g. 4|03 (<100 proc's)
- last agreed seq.no was 6|05
- two proc's p_3, p_5 receive multicasts in different order from p_1, p_8





Performance?
- 3 serial messages!

Total Multicast (cont.)

```
Data structures at process p:
    $A_g^p$ : largest agreed seq.no. seen,  $P_g^p$ : largest proposed seq.no. of process p
On initialization:  $P_g^p = 0$
For process p to **TO-multicast** message m to group g
    **B-multicast ( g, <m,  i >)**     /i = unique id for m, contains sender no.
On **B-deliver (<m, i> )** at q from p
    $P_g^q = \max(A_g^q, P_g^q) + 1$; send (p, < i, $P_g^q$ >);
  store <m, i, $P_g^q$ > in hold-back Q: for order, use $A_g^q$ if assigned, $P_g^q$ otherwise
On receive (<i, P>)  at p from q (wait for all replies; A is the largest reply)
    **B-multicast(g, < "order", i, A >)**
On **B-deliver (<"order", i, A > )** at q from p
    $A_g^q = \max(A_g^q, A)$
    attach A to message i  in hold-back Q → re-order messages in hold-back Q,
            (increasing seq.no.'s, if identical seq.no.: use sender no. to order)
    while msg m in front of hold-back Q has been assigned an agreed seq.no.
            do          remove m from hold-back Q; **TO-deliver (m)**     od;
```

Consensus: Introduction

Some consensus problems

Not used in 2015/16 Lecture

„all correct computers controlling a spaceship should decide to proceed with landing, or all of them should decide to abort (after each has proposed one action or the other)“

„in an electronic money transfer transaction, all involved processes must consistently agree on whether to perform the transaction (debit and credit), or not“

„in mutual exclusion, processes need to agree on which process enters critical section“

„in election, processes need to agree on elected process“

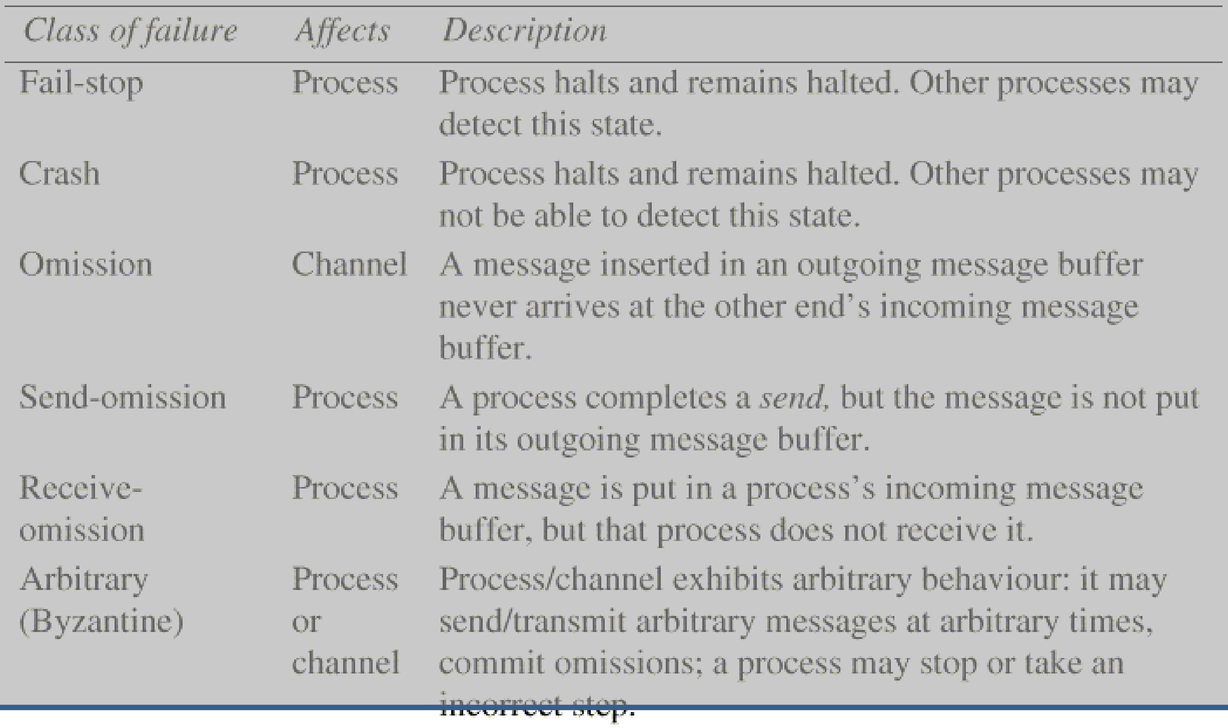
„in totally ordered multicast, processes need to agree on a consistent message delivery order“

Consensus: Introduction (cont.)

Not used in 2015/16 Lecture

process failure models

- crash failures: processes stop (fail), but remain silent
- byzantine failures: processes fail, but may still respond to environment with arbitrary, erratic behavior (e.g., send false acknowledgements, etc.)
- ... a more detailed definition (for your records):



Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send,* but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Consensus: Introduction (cont.)

Factors threatening consensus: Not used in 2015/16 Lecture

- failures
  - communication link or process failures
  - crash failures (fail-silent) or byzantine failures (arbitrary)
    (after Byzantine Empire 330-1453, in which unfaithfulness and untruthfulness have allegedly been very common)
- network characteristics
  - synchronous or asynchronous
- failure detectors
  - reliable or unreliable
- are msgs authenticated (digitally signed) or not
  - can a process lie about the content of message that it received from a correct process?
  - can adversary claim to send message under a false expedient's ID?

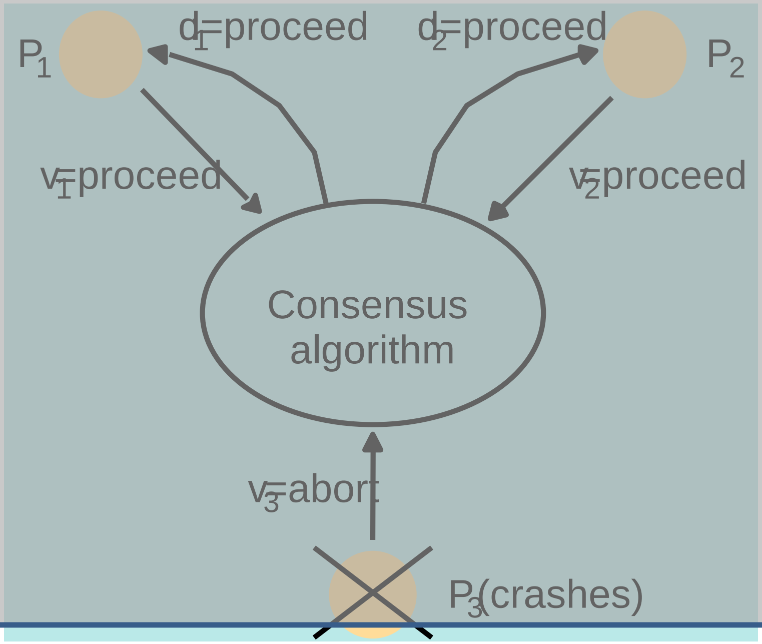
Model:

- processes communicating by msg passing
- desirable: reaching consensus even in the presence of faults
  - assumption: communication is reliable, but processes may fail

Consensus: C

The Consensus Problem (C): Not used in 2015/16 Lecture

- agreement wrt. value of a decision variable among all correct proc's
  - p_i is in state *undecided* & proposes a single value v_i
  - next, processes communicate with each other to exchange values
  - in doing so, p_i sets decision variable d_i and enters *decided* state after which the value of d_i remains unchanged



Consensus: BG

Not used in 2015/16 Lecture

The Byzantine Generals Problem (BG)

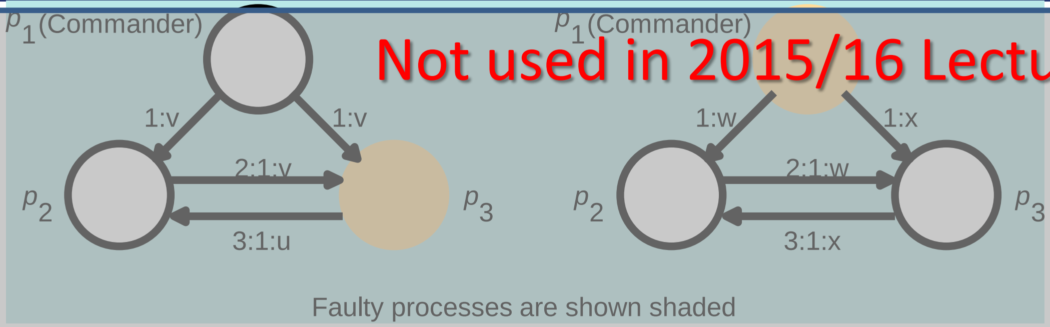
- ≥ 3 generals to agree on attack or retreat; commander issues order, others (lieutenant to commander) must decide: attack or retreat
- one of the generals may be treacherous
  - if commander treacherous:
    proposes attacking to one general and retreating to the other
  - if lieutenants treacherous: tell some peers that commander ordered to attack, and others that commander ordered to retreat
- difference to consensus problem:
  one process supplies a value that others have to agree on
- properties
  - termination: eventually each correct process sets decision variable
  - agreement: the decision value of all correct processes is the same
  - integrity: if commander is correct, then all processes decide on the value that the commander proposes
    - note: implies agreement only if commander is correct, but:
      commander need not be correct (see above)

Consensus: BG in Sync. Net

Not used in 2015/16 Lecture

- allow arbitrary (byzantine) failures
- up to f faulty processes
- correct processes can detect absence of a msg through timeout, but: cannot conclude that sender crashed, since it may be silent for some time and then start sending messages again
- assume private communication channels
  - 4th proc cannot detect
    if proc sends msg's w/ different content to 2 peers
  - no faulty proc can inject msgs into channels connecting correct proc's
- assume that msgs are not digitally signed (authenticated and verifiable)
- general result (Lamport, Shostak and Pease)
  - no solution if $N \leq 3f$
  - give an algorithm for $N \geq 3f+1$

Consensus: BG in Sync. Net (cont.)



Faulty processes are shown shaded

Not used in 2015/16 Lecture

impossibility for N = 3 processes

- read “3:1:u” as “three says one says u”
- both scenarios show two rounds of messages
- left: all p_2 knows is that it has received two different values
- right: same situation, even though now commander is faulty
- assume a solution existed
  - p_2 would have to decide on value v, by integrity condition of BG
- assume no algorithm can decide locally for p_2 between the 2 scenarios
  - then p_2 would need to decide on w (sent by commander) in 2nd scenario
- same reasoning for p_3 w
  - will have to decide about commander’s value, which is violation of agreement in right hand scenario, hence contradiction

Consensus: BG in Sync. Net (cont.)

TECHNISCHE UNIVERSITÄT DARMSTADT

impossibility for N $\leq$ 3f (Pease, Shostak & Lamport; sketch)

Not used in 2015/16 Lecture

- assume a solution existed for N $\leq$ 3f → construct solution for (3, 1)
- let each of 3 proc‘s p_1, p_2 & p_3 simulate n_1, n_2 & n_3 generals, where $n_1+n_2+n_3 = N$ and $n_1, n_2, n_3 \leq f$
- assume that one of the processes is faulty
- correct processes simulate correct generals
  - internal interaction of “own” generals
  - send msgs from “own” generals to those simulated by other processes
- faulty general’s f proc‘s are faulty and may emit spurious messages
- since $n_1+n_2+n_3 = N$ and $n_1, n_2, n_3 \leq N/3$, $\leq$ f generals are faulty
- however, now there is a way for 2of3 proc‘s to reach consensus: each proc decides on value chosen by all generals it simulates
- contradicts impossibility for N = 3

Consensus: BG in Sync. Net (cont.)

solution for N ≥ 3f+1 Not used in 2015/16 Lecture

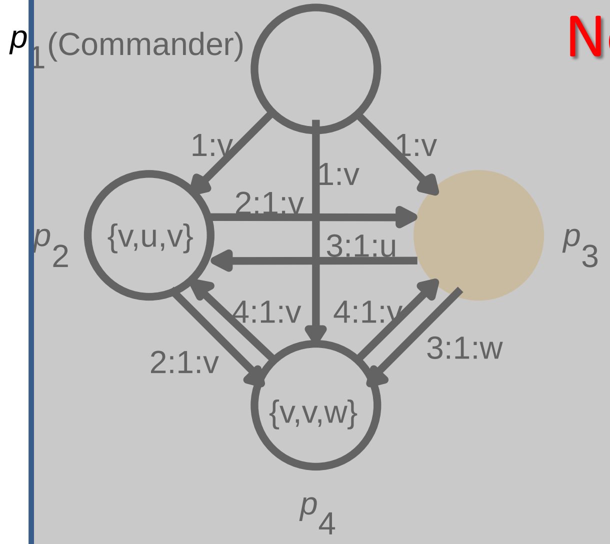
- solution by Pease, Shostak & Lamport too complex to present (cf. http://www.cs.wisc.edu/~bart/739/papers/byzantine.pdf), therefore:

presentation of solution for N = 4, f = 1

- correct generals reach agreement in two rounds:
  - first, commander sends value to each lieutenant
  - second, each lieutenant sends value it received to all peers
- lieutenant receives:
  1 value from commander, N-2 values from peers
- if commander faulty, then all lieutenants correct, each will have gathered exactly the set of values that the commander sent out
- if one lieutenant faulty, each of its peers receives N-2 copies of the value the commander sent out, plus the faulty lieutenant value
- to reach agreement, simple majority function suffices
  - since N ≥ 4, N-2 ≥ 2, majority function will ignore value of faulty lieutenant, and produce value of commander if commander is correct (may produce ⊥ if commander incorrect)
- note: BG requires integrity only if commander correct

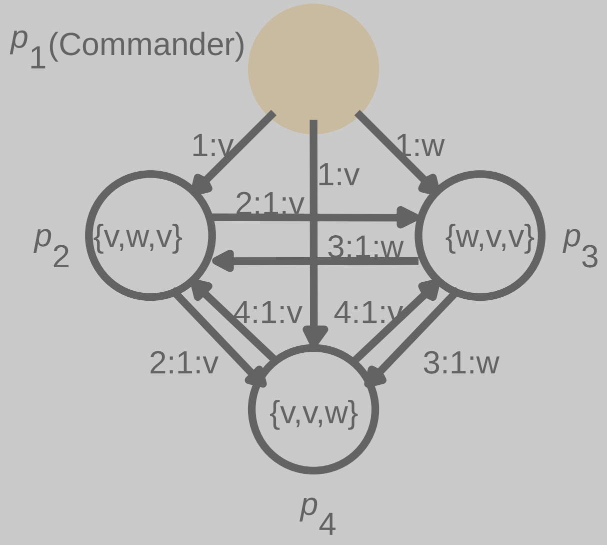
Consensus: BG in Sync. Net (cont.)

Not used in 2015/16 Lecture



p_2: majority({v,u,v}) = v

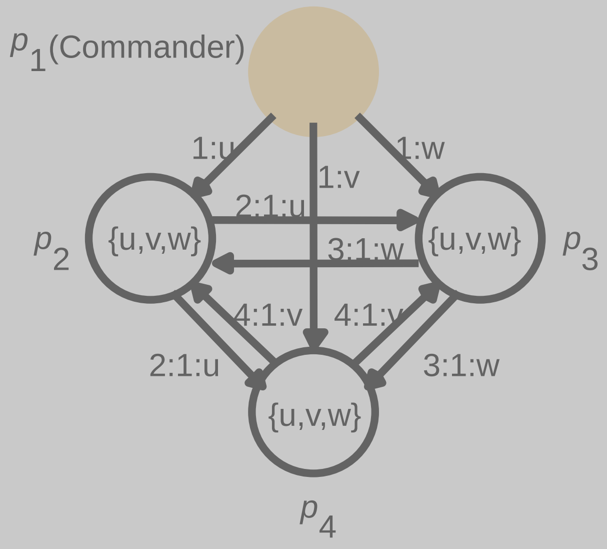
p_3: majority({v,v,w}) = v



p_2: majority({v,w,v}) = v

p_3: majority({v,v,w}) = v

p_4: majority({w,v,v}) = v



p_2, p_3, p_4:
majority({v,u,w}) = $\perp$

Consensus: Asynchronous Systems

Impossibility of agreement in asynchronous systems

Not used in 2015/16 Lecture

- previous algo.s: synchrony assumed (msg exchange in rounds, timeouts)
- in asy. systems, no algo. can guarantee reaching consensus, even with just one process crash failure (Fischer, Lynch and Paterson, 1985)
  - proof idea: show that there is always some continuation of the proc's execution that avoids consensus being reached
- consequences: in async. systems, no solution to BG, IC, TOR-multicast
- of course, in practice consensus can often be reached, but a residual probability that consensus cannot be reached remains
- possible approaches to reaching consensus by weakening assumptions:
  - partial synchrony
  - masking faults
  - modified failure detectors
  - randomized algorithms

Books

- G. Coulouris, J. Dollimore, T. Kindberg, Verteilte Systeme - Konzepte und Design 3. Aufl (2002) Pearson Studium
  - original: english; both available & good
  - closest to chapter 2, some slides included
- A. Tanenbaum, M.v.Steen, Verteilte Systeme: Grundlagen und Paradigmen, Pearson Studium 2003
  - original: english; both available & good
  - yet, MM: „does not compare to Tanenbaum‘s ‚networks‘ or to ‚Coulouris‘“
- G. Tel, Introduction to Distributed Algorithms 2nd Ed (2001); Cambridge University Press, ISBN: 05217948
  - english, available
  - for those who want to go deep!