Application-Level Checkpoint-restart (CPR) for MPI Programs

Keshav Pingali

Joint work with Dan Marques, Greg Bronevetsky, Paul Stodghill, Rohit Fernandes

The Problem

- Old picture of high-performance computing:
  - Turn-key big-iron platforms
  - Short-running codes
- Modern high-performance computing:
  - Roll-your-own platforms
    - Large clusters from commodity parts
    - Grid Computing
  - Long-running codes
    - Protein-folding on BG may take 1 year
- Program runtimes are exceeding MTBF
  - ASCI, Blue Gene, Illinois Rocket Center

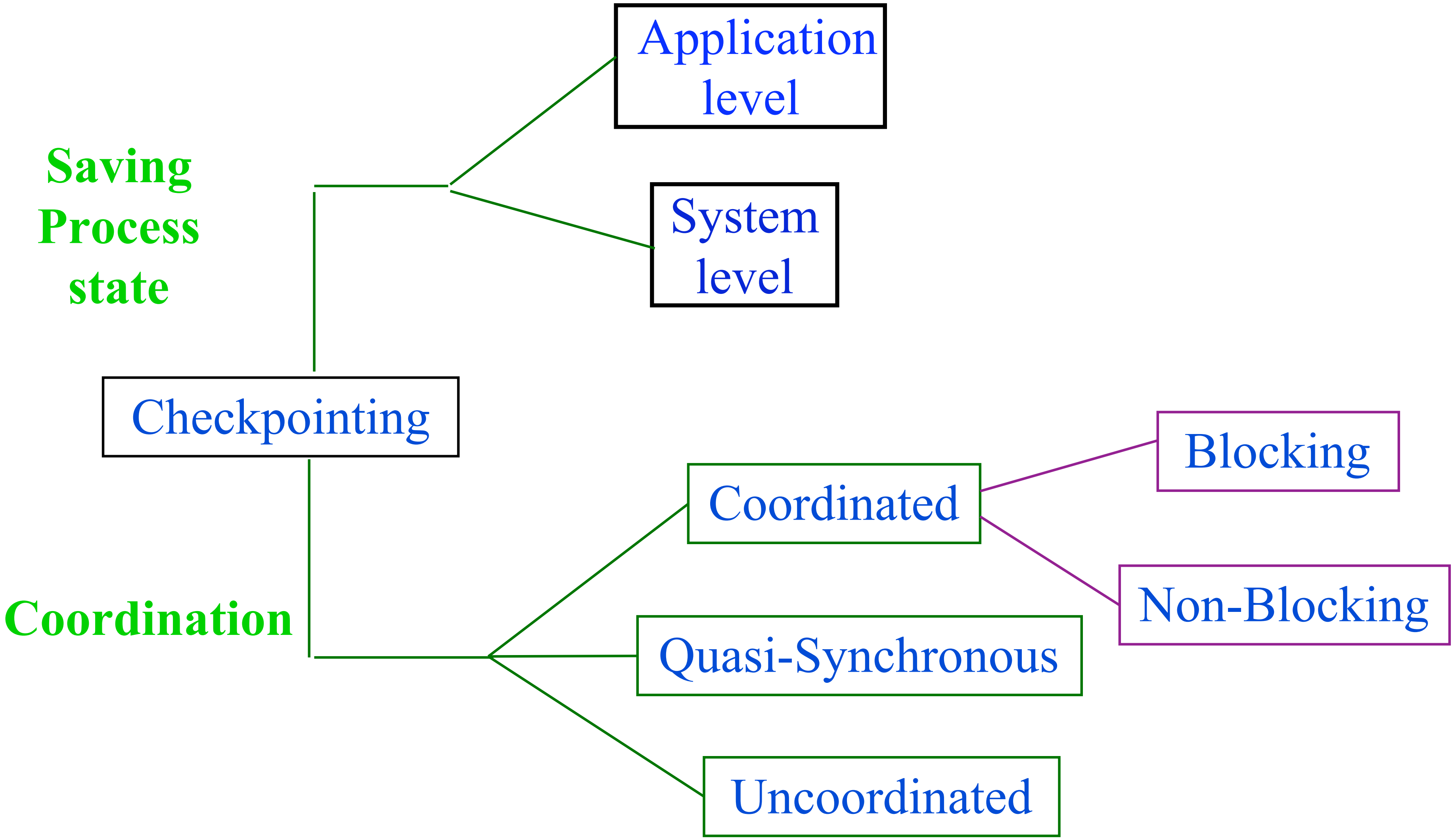
Software view of hardware failures

- Two classes of faults
  - Fail-stop: a failed processor ceases all operation and does not further corrupt system state
  - Byzantine: arbitrary failures
    - Nothing to do with adversaries
- Our focus:
  - Fail-Stop Faults

Solution Space for Fail-stop Faults

- Checkpoint-restart (CPR) *[Our Choice]*
  - Save application state periodically
  - When a process fails, all processes go back to last consistent saved state.
- Message Logging
  - Processes save outgoing messages
  - If a process goes down it restarts and neighbors resend it old messages
  - Checkpointing used to trim message log
  - In principle, only failed processes need to be restarted
  - Popular in the distributed system community
  - Our experience: not practical for scientific programs because of communication volume

Solution Space for CPR



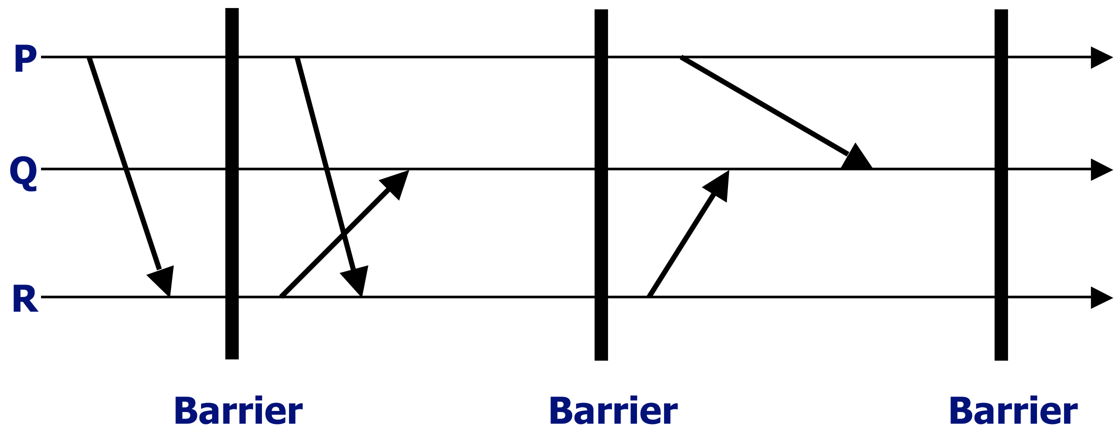
Saving process state

- System-level (SLC)
  - save all bits of machine
  - program must be restarted on same platform
- Application-level (ALC) *[Our Choice]*
  - programmer chooses certain points in program to save minimal state
  - programmer or compiler generate save/restore code
  - amount of saved data can be much less than in system-level CPR (e.g., n-body codes)
  - in principle, program can be restarted on a totally different platform
- Practice at National Labs
  - demand vendor provide SLC
  - but use hand-rolled ALC in practice!

Coordinating checkpoints

- Uncoordinated
  - Dependency-tracking, time-coordinated, …
  - Suffer from exponential rollback
- Coordinated *[Our Choice]*
  - Blocking
    - Global snapshot at a Barrier
    - Used in current ALC implementations
  - Non-blocking
    - Chandy-Lamport

Blocking Co-ordinated Checkpointing

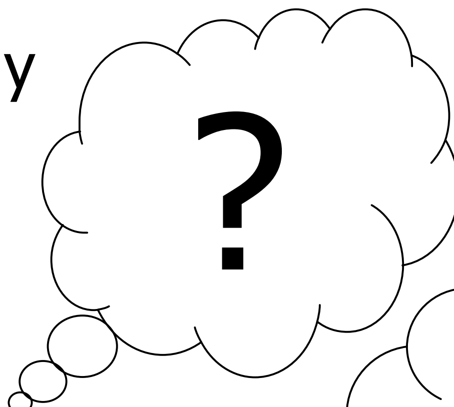


- Many programs are bulk-synchronous (BSP model of Valiant)
- At barrier, all processes can take checkpoints.
  - assumption: no messages are in-flight across the barrier
- Parallel program reduces to sequential state saving problem
- But many new parallel programs do not have global barriers..

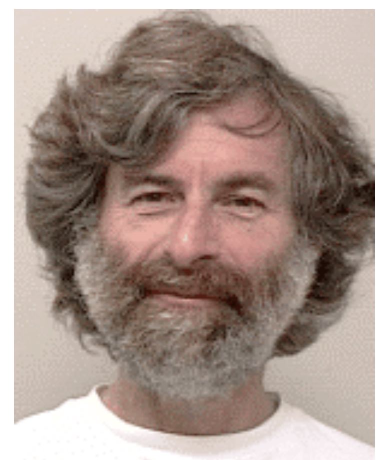
Non-blocking coordinated checkpointing

- Processes must be coordinated, but …
- Do we really need to block all processes before taking a global checkpoint?

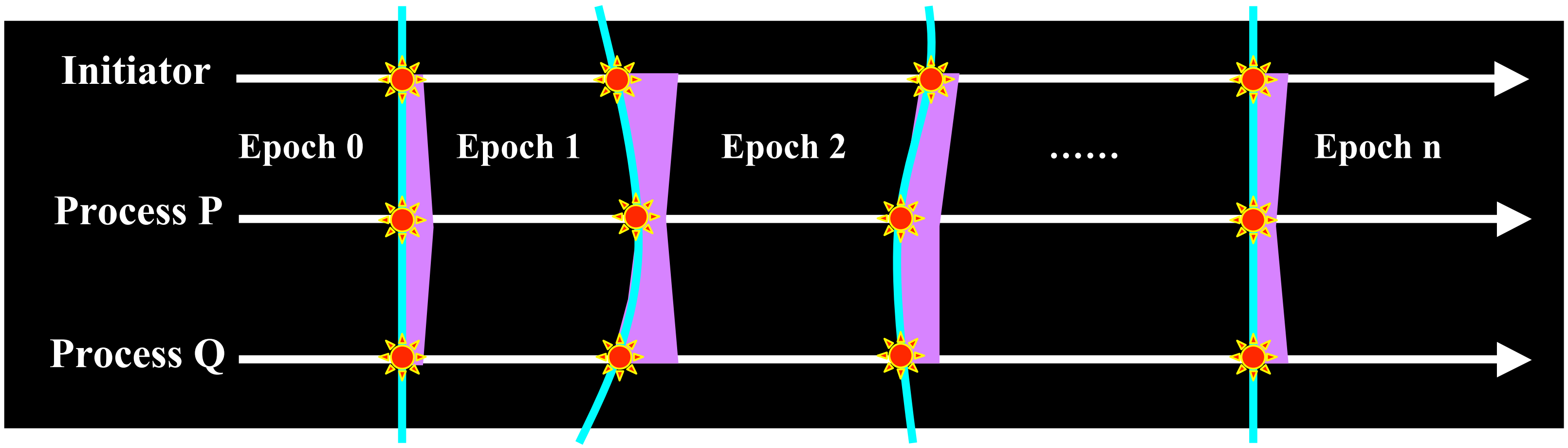
K. Mani Chandy



Leslie Lamport

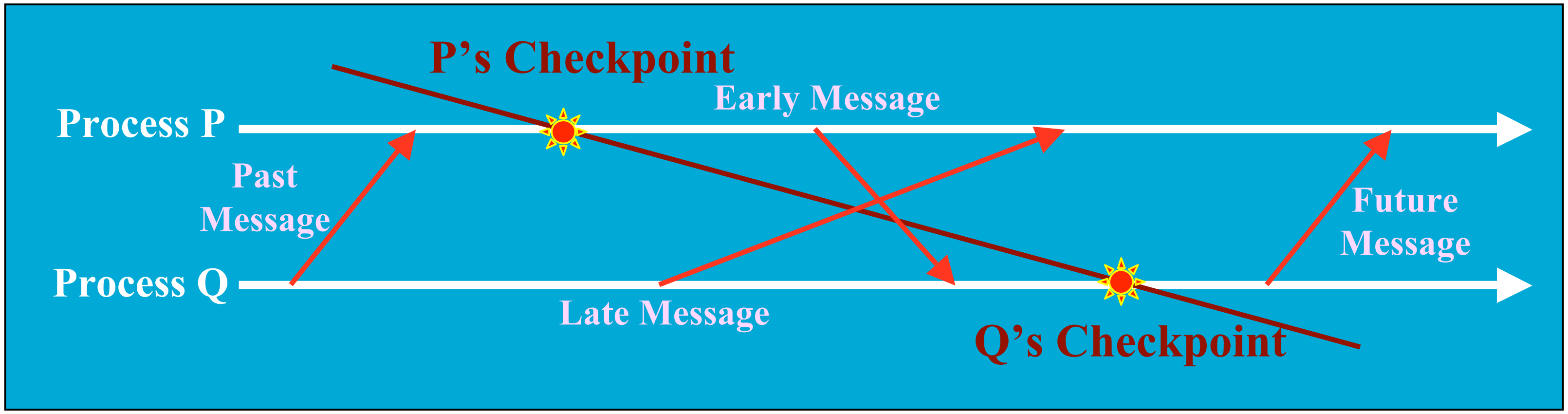


Global View



- Initiator
  - root process that decided to take a global checkpoint once in a while
- Recovery line
  - saved state of each process (+ some additional information)
  - recovery lines do not cross
- Epoch
  - interval between successive recovery lines
- Program execution is divided into a series of disjoint epochs
- A failure in epoch n requires that all processes roll back to the recovery line that began epoch n

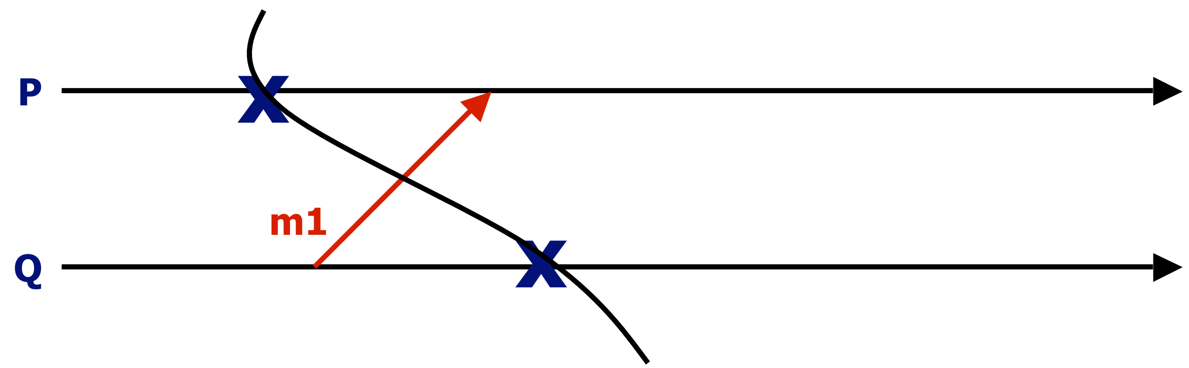
Possible Types of Messages



- On Recovery:
  - Past message will be left alone.
  - Future message will be reexecuted.
  - Late message will be re-received but not resent.
  - Early message will be resent but not re-received.

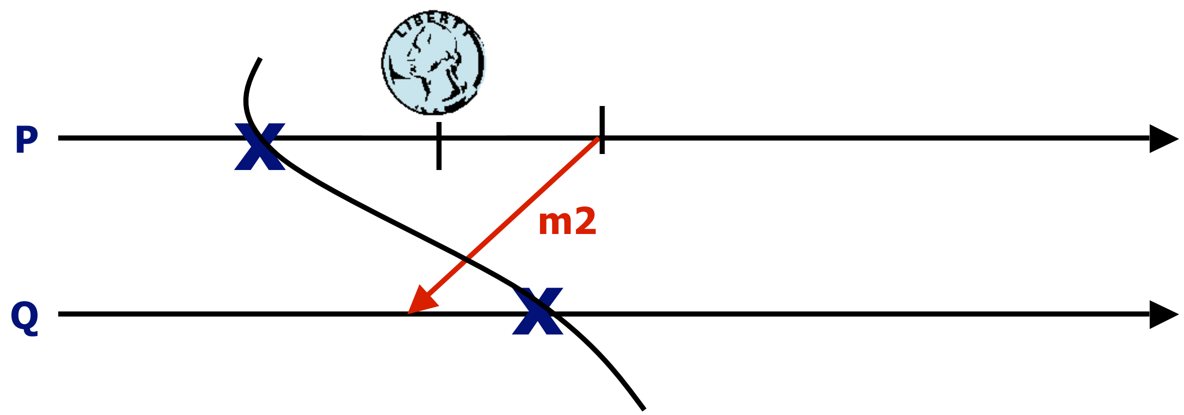
➔ Non-blocking protocols must deal with late and early messages.

Difficulties in recovery: (I)



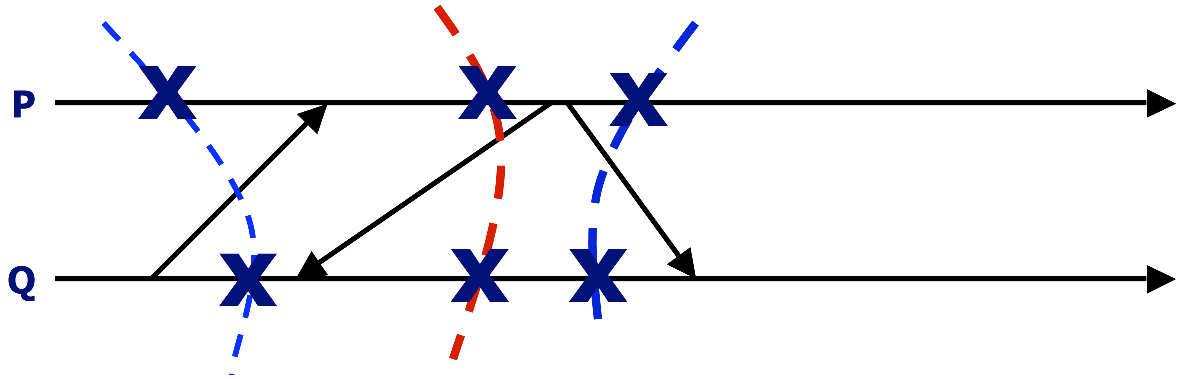
- Late message: m1
  - Q sent it before taking checkpoint
  - P receives it after taking checkpoint
- Called *in-flight* message in literature
- On recovery, how does P re-obtain message?

Difficulties in recovery: (II)



- Early message: m2
  - P sent it after taking checkpoint
  - Q receives it before taking checkpoint
- Called inconsistent message in literature
- Two problems:
  - How do we prevent m2 from being re-sent?
  - How do we ensure non-deterministic events in P relevant to m2 are re-played identically on recovery?

Approach in systems community



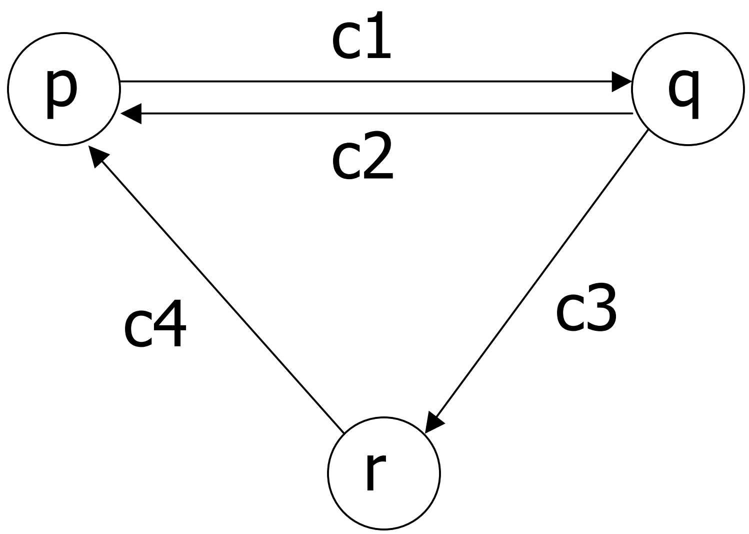
- Ensure we never have to worry about inconsistent messages during recovery
- Consistent cut:
  - Set of saved states, one per process
  - No inconsistent message

➔ saved states must form a consistent cut

- Ensuring this: Chandy-Lamport protocol

Chandy-Lamport protocol

- Processes
  - one process initiates taking of global snapshot
- Channels:
  - directed
  - FIFO
  - reliable
- Process graph:
  - Fixed topology
  - Strongly connected component



Algorithm explanation

1. Coordinating process state-saving
   – How do we avoid inconsistent messages?
2. Saving in-flight messages
3. Termination

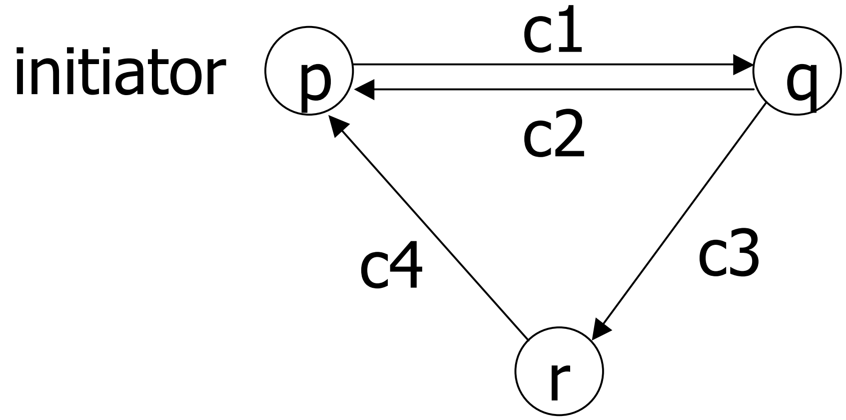
Next: Model of Distributed System

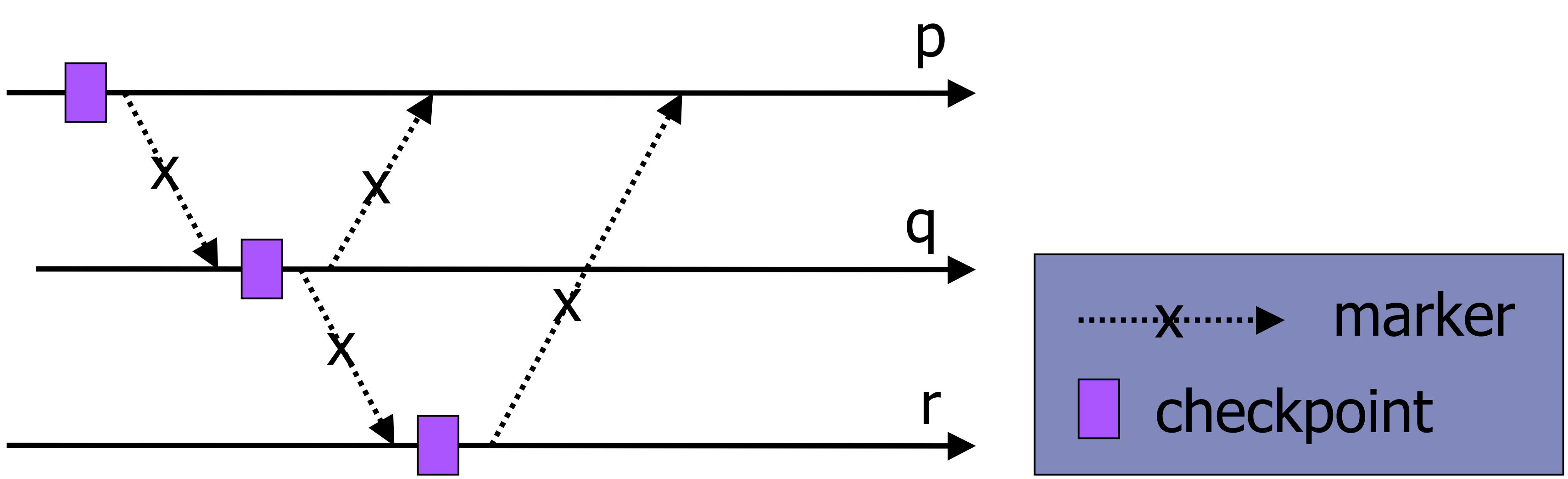
Step 1: co-ordinating process state-saving

- Initiator:
  - Save its local state
  - Send a *marker token* on each outgoing edge
    - Out-of-band (non-application) message
- All other processes:
  - On receiving a marker on an incoming edge for the first time
    - save state immediately
    - propagate markers on all outgoing edges
    - resume execution.
  - Further markers will be eaten up.

Next: Example

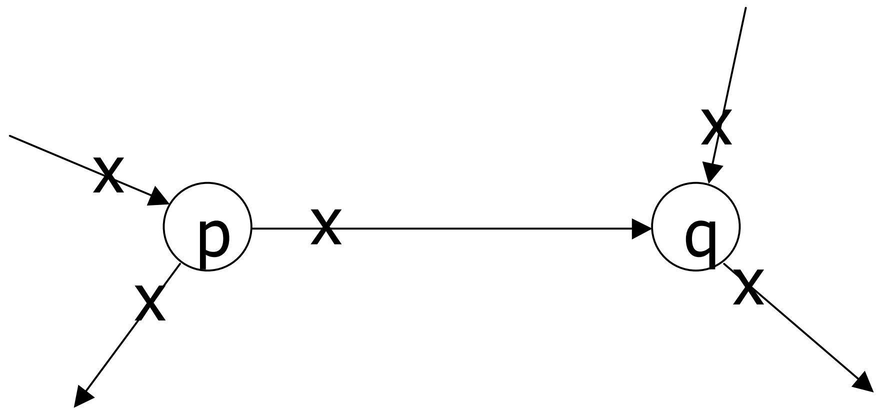
Example



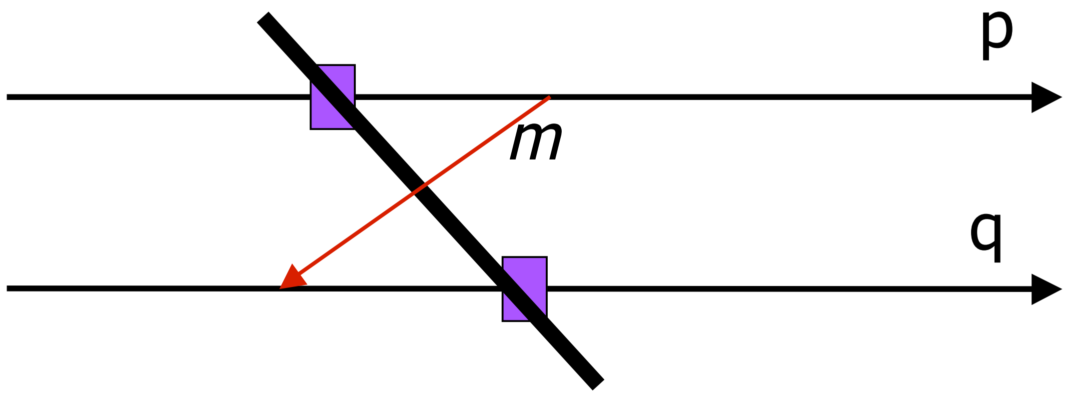


Next: Proof

Theorem: Saved states form consistent cut

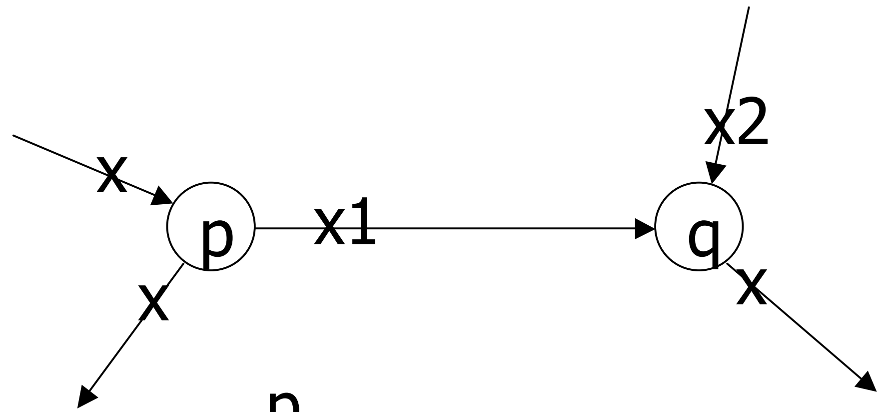


Let us assume that a message *m* exists, and it makes our cut inconsistent.

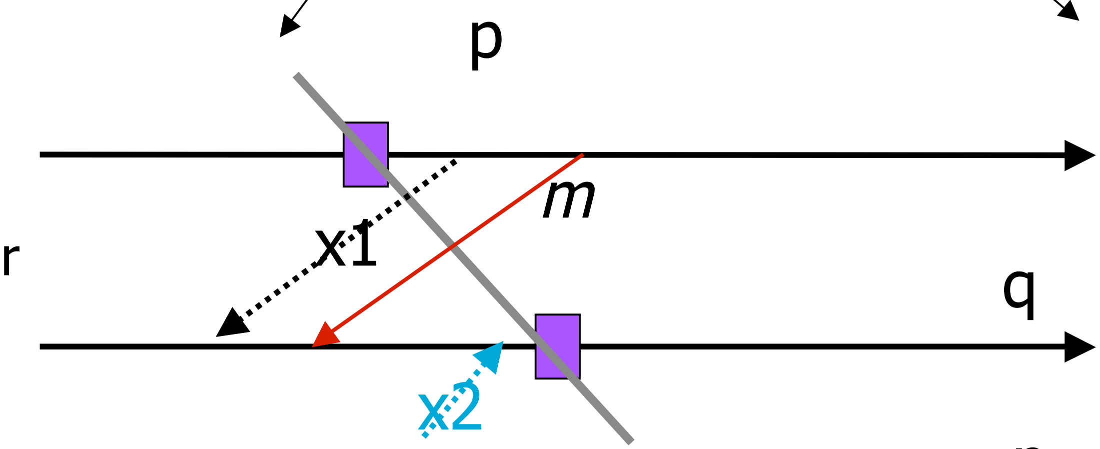


Next: Proof (cont')

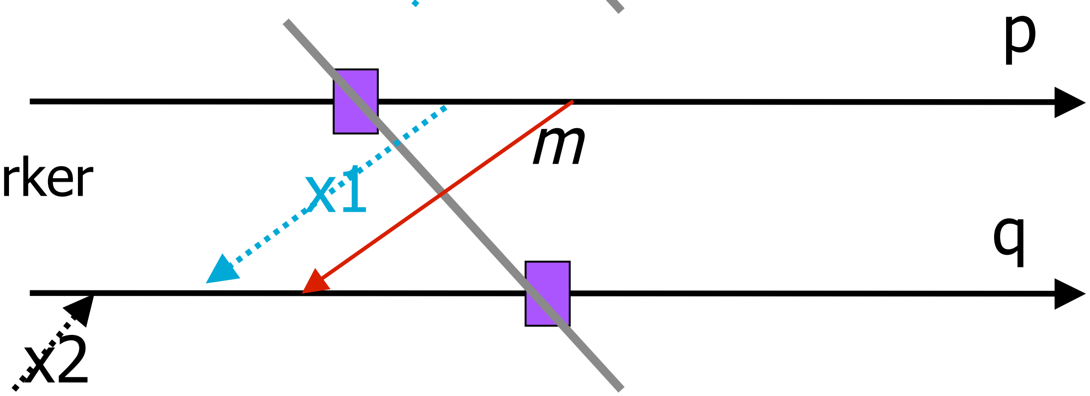
- Proof(cont')



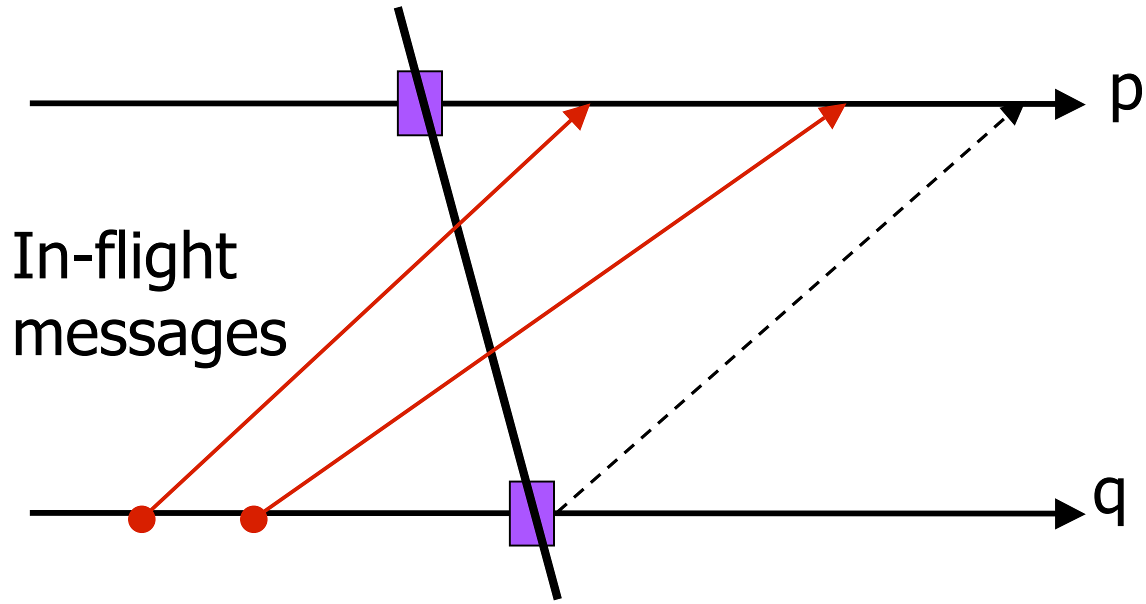
(1) x1 is the 1st marker
for process q



(2) x1 is not the 1st marker
for process q



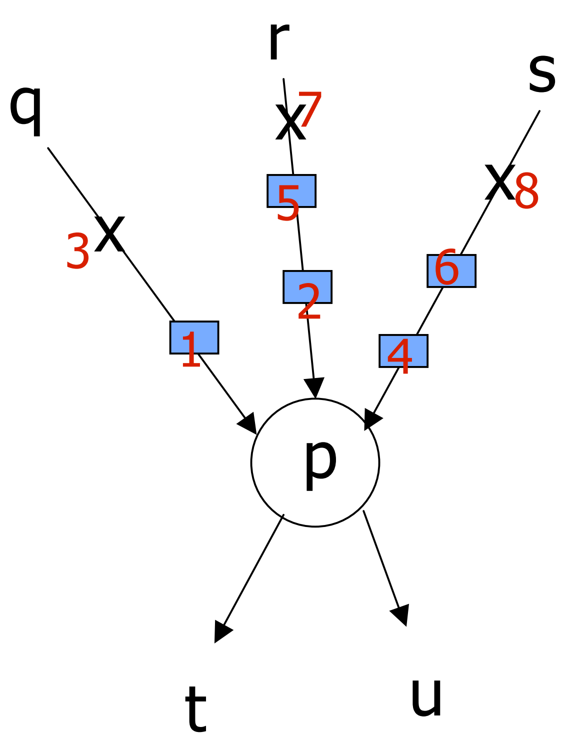
Step 2:recording in-flight messages



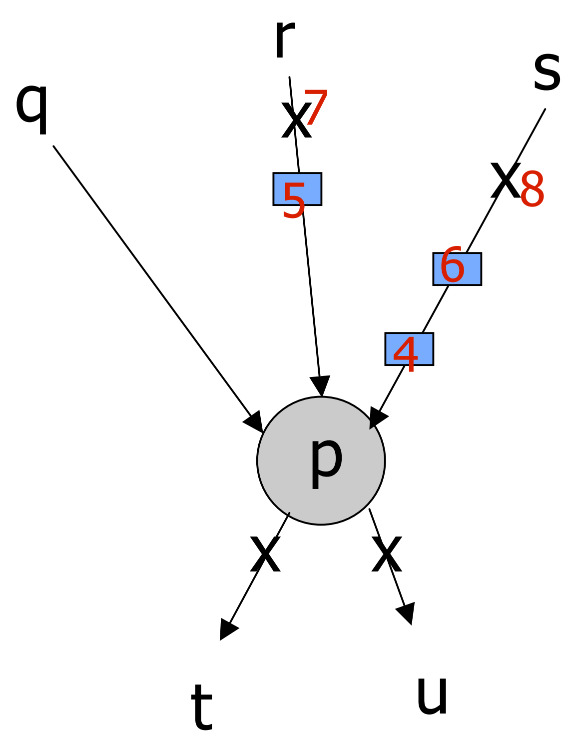
- Process p saves all messages on channel c that are received
  - after p takes its own checkpoint
  - but before p receives marker token on channel c

Example

(1) p is receiving messages

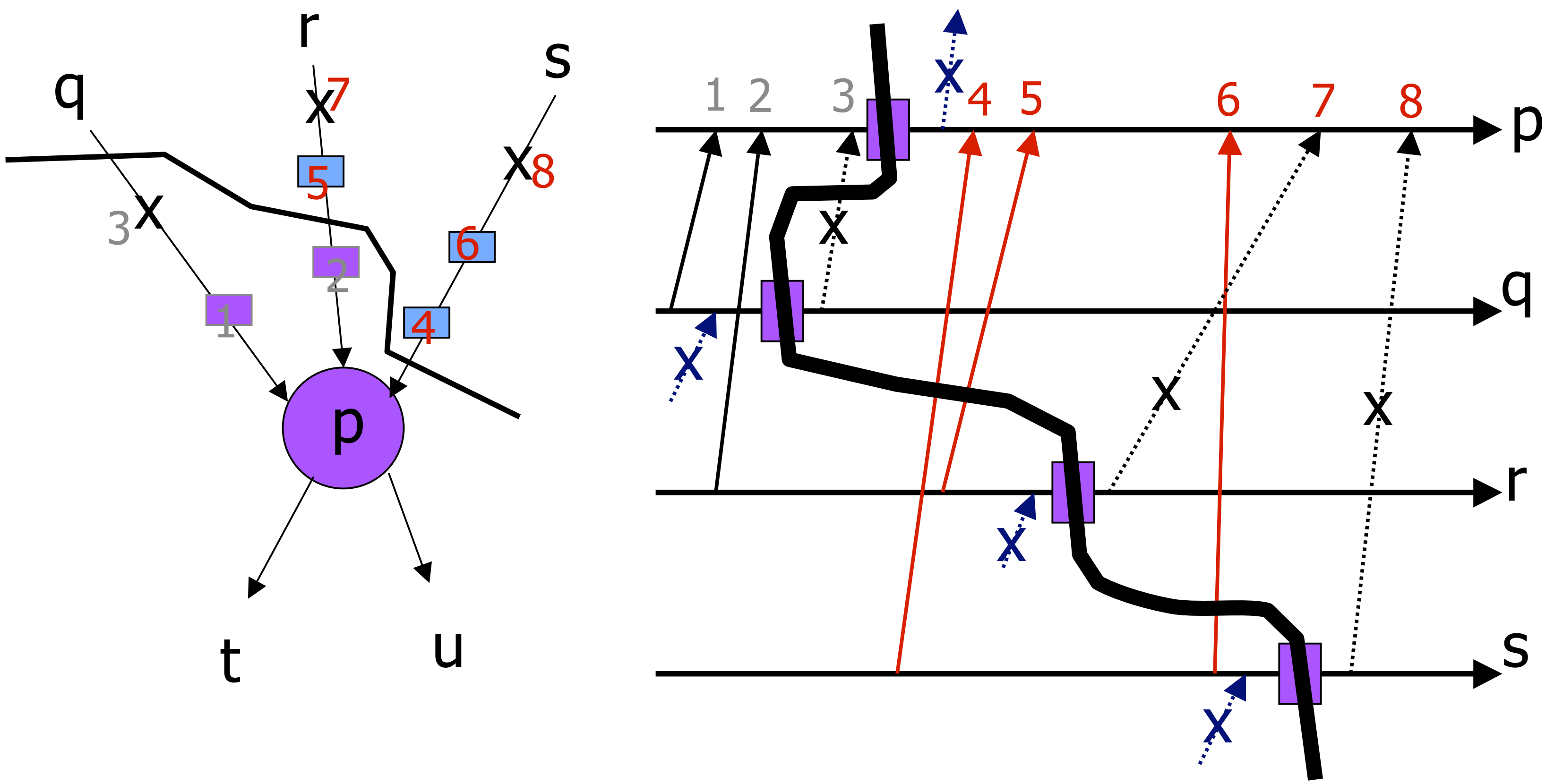


(2) p has just saved its state



Example(cont')

p's chkpnt triggered by a marker from q



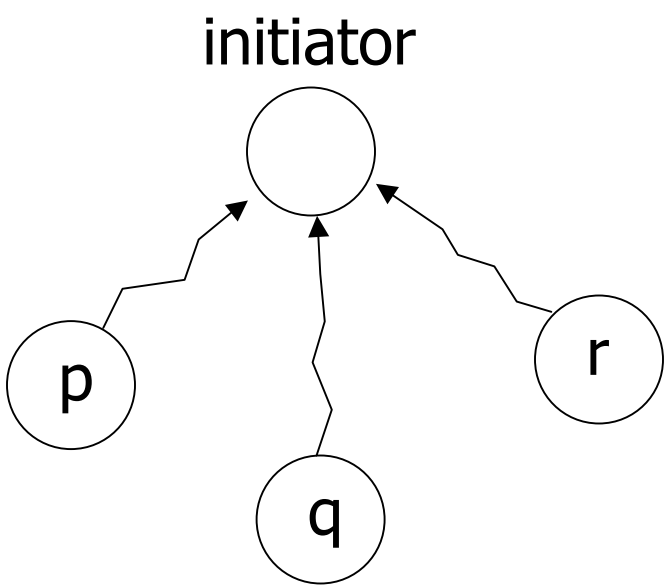
Next: Algorithm (revised)

Algorithm (revised)

- Initiator: when it is time to checkpoint
  - Save its local state
  - Send marker tokens on all outgoing edges
  - Resume execution, but also record incoming messages on e ach in-channel c until marker arrives on channel c
  - Once markers are received on all in-channels, save in-flight messages on disk
- Every other process: when it sees first marker on any in-channel
  - Save state
  - Send marker tokens on all outgoing edges
  - Resume execution, but also record incoming messages on e ach in-channel c until marker arrives on channel c
  - Once markers are received on all in-channels, save in-flight messages on disk

Step 3: Termination of algorithm

- Did every process save its state and its in-flight messages?
  - outside scope of C-L paper



- direct channel to the initiator?
- spanning tree?

Comments on C-L protocol

- Relied critically on some assumptions:
  - Process can take checkpoint at any time during execution
    - get first marker → save state
  - FIFO communication
  - Fixed communication topology
  - Point-to-point communication: no group communication primitives like bcast
- None of these assumptions are valid for application-level checkpointing of MPI programs

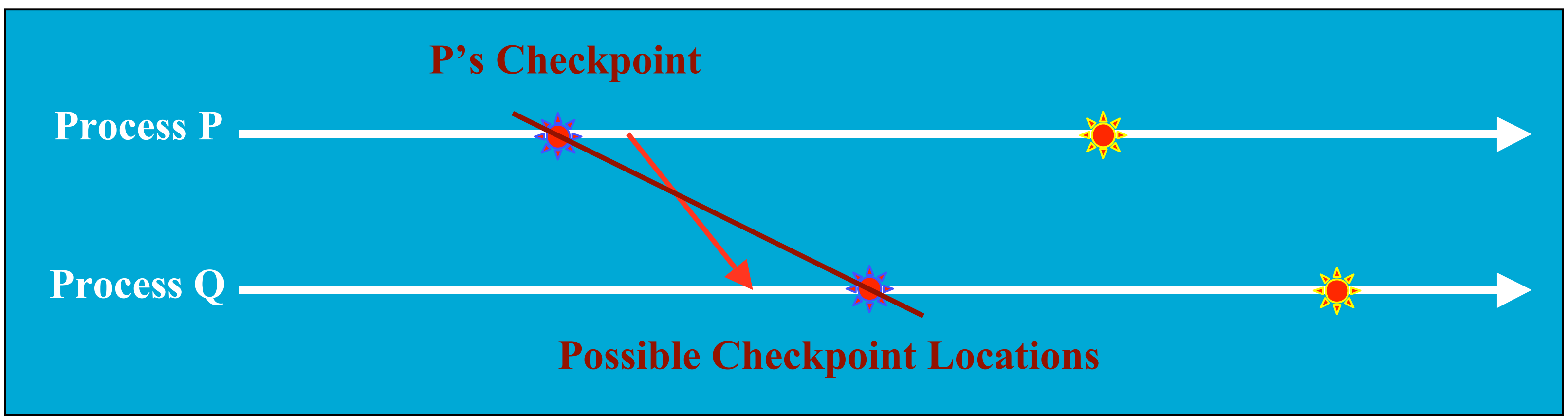
Application-Level Checkpointing (ALC)

- At special points in application the programmer (or automated tool) places calls to a *take_checkpoint()* function.
- Checkpoints may be taken at such spots.
- State-saving:
  - Programmer writes code
  - Preprocessor transforms program into a version that saves its own state during calls to *take_checkpoint()*.

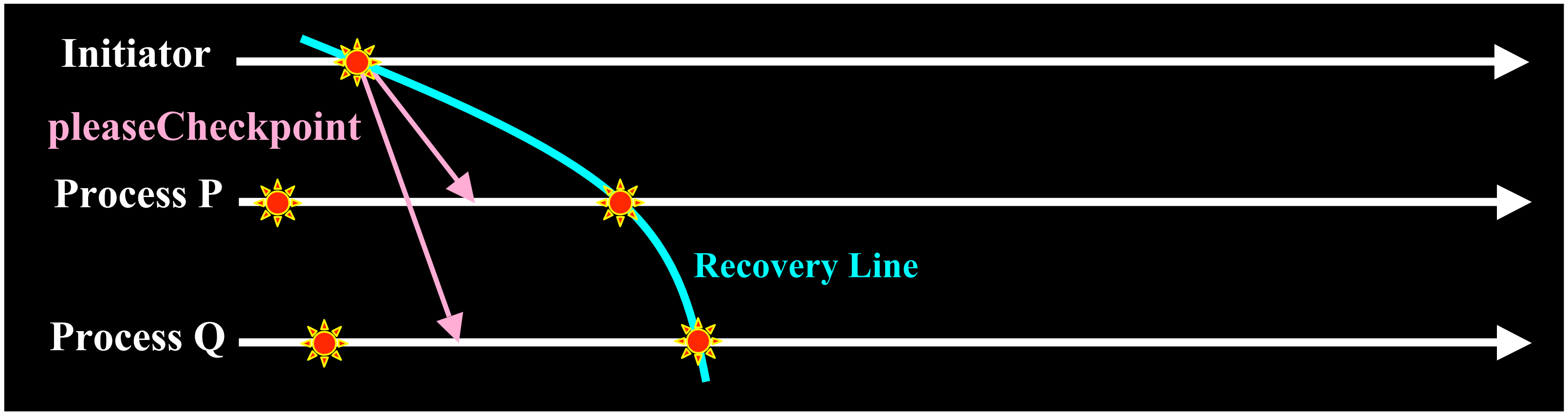
Application-level checkpointing difficulties

- System-level checkpoints can be taken anywhere
- Application-level checkpoints can only be taken at certain places in program
- This may lead to inconsistent messages

➔ Recovery lines in ALC may form inconsistent cuts

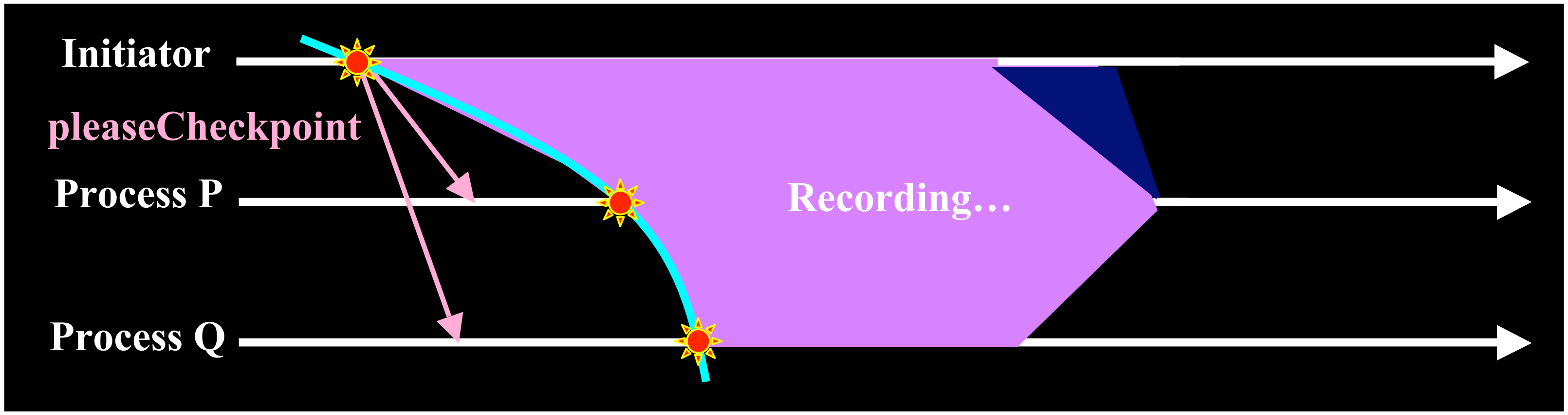


Our protocol (I)



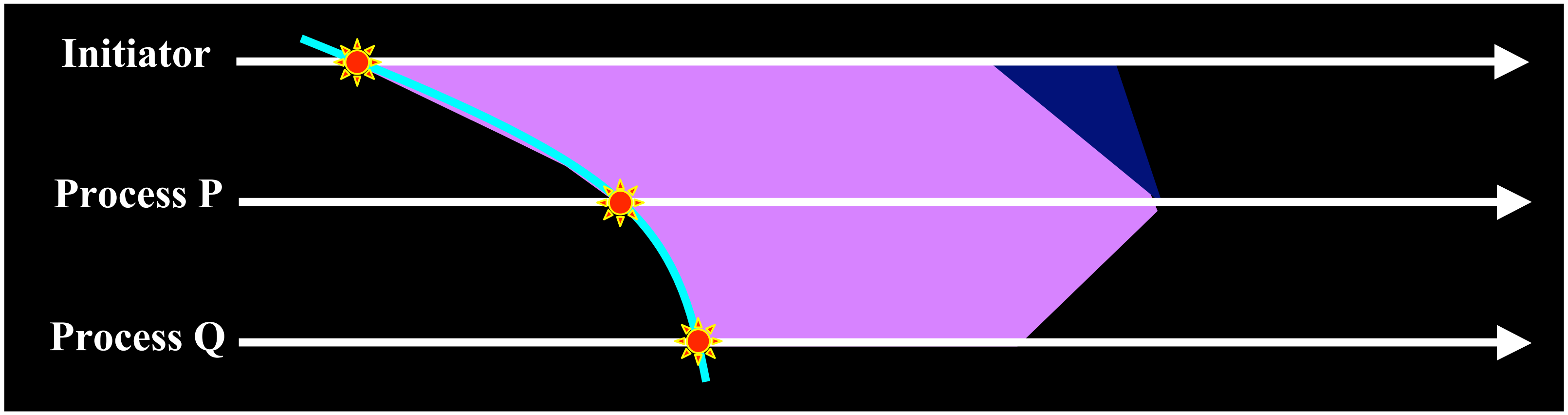
- Initiator checkpoints, sends *pleaseCheckpoint* message to all others
- After receiving this message, process checkpoints at the next available spot
  - Sends every other process Q the number of messages sent to Q in the last epoch

Protocol Outline (II)



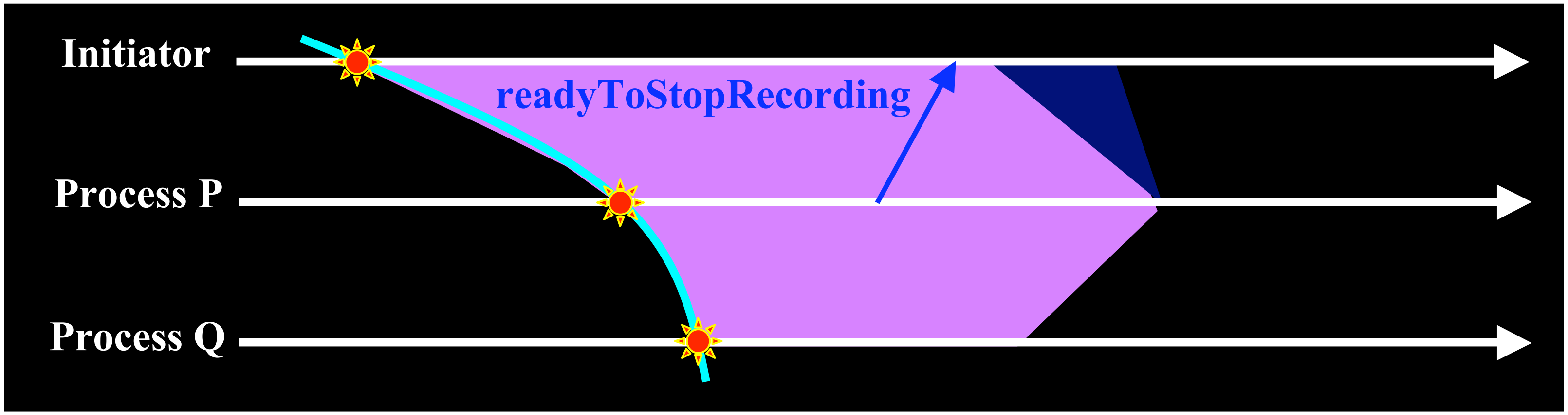
- After checkpointing, each process keeps a record, containing:
  - data of messages from last epoch (Late messages)
  - non-deterministic events:
    - In our applications, non-determinism arises from wild-card MPI receives

Protocol Outline (IIIa)



- Globally, ready to stop recording when
  - all processes have received their late messages
  - no process can send early message
    - safe approximation: all processes have taken their checkpoints

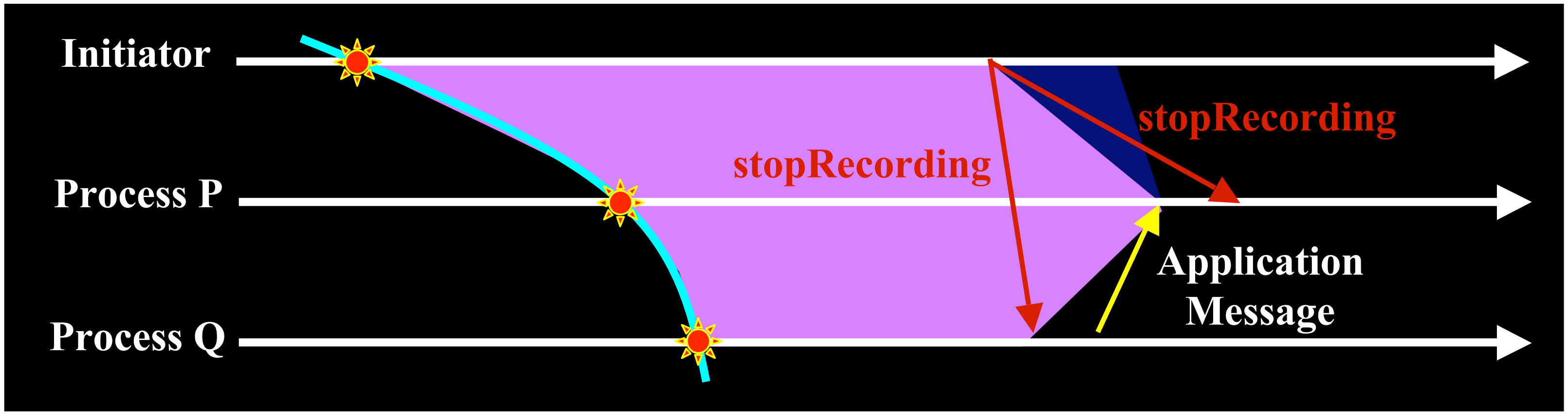
Protocol Outline (IIIb)



- Locally, when a process
  - has received all its late messages

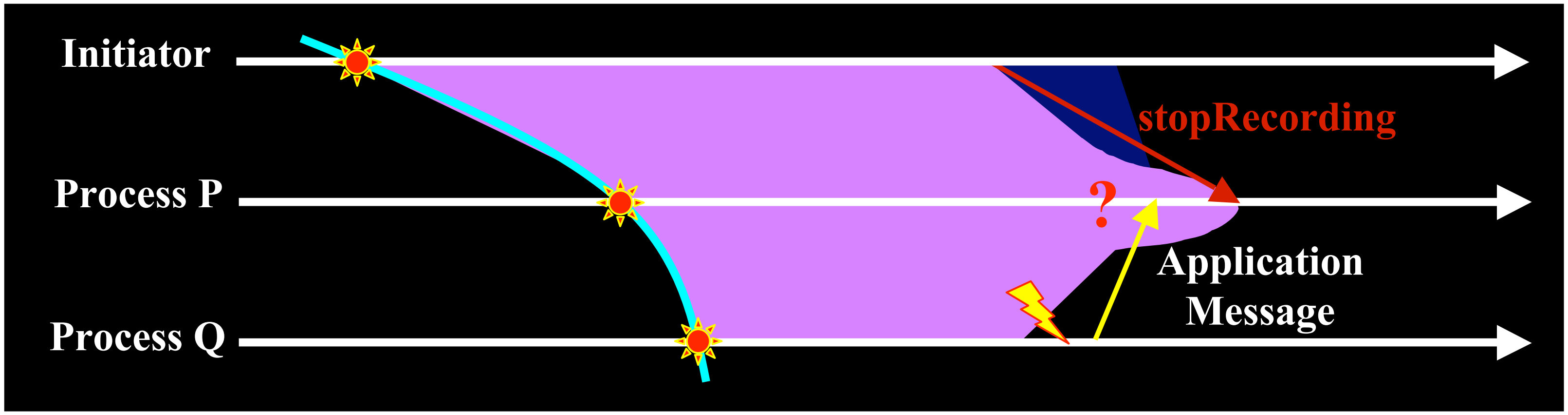
$\Rightarrow$ sends a *readyToStopRecording* message to Initiator.

Protocol Outline (IV)



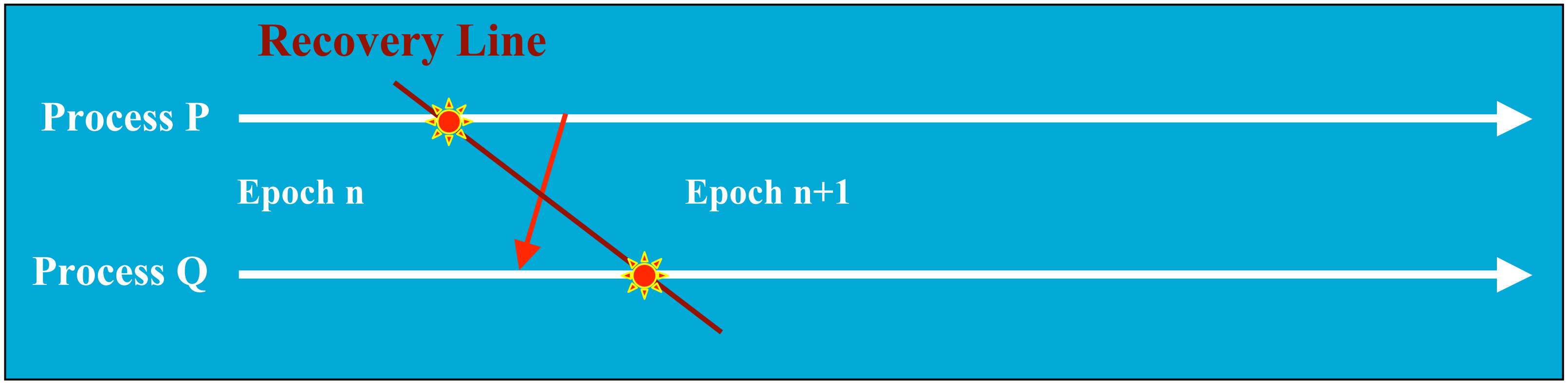
- When initiator receives *readyToStopRecording* from everyone, it sends *stopRecording to everyone*
- Process stops recording when it receives
  - *stopRecording* message from initiator OR
  - message from a process that has itself stopped recording

Protocol Discussion



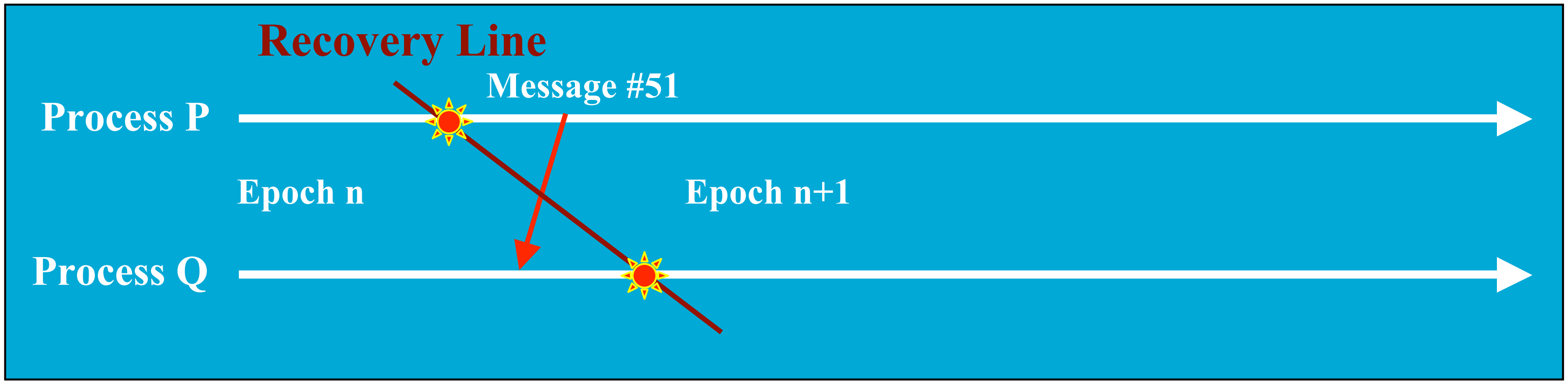
- Why can't we just wait to receive *stopRecording* message?
- Our record would depend on a non-deterministic event, invalidating it.
  - The application message may be different or may not be resent on recovery.

Non-FIFO channels



- In principle, we can piggyback epoch number of sender on each message
- Receiver classifies message as follows:
  - Piggybacked epoch < receiver epoch: late
  - Piggybacked epoch = receiver epoch: intra-epoch
  - Piggybacked epoch > receiver epoch: early

Non-FIFO channels



- We can reduce this to one bit:
  - Epoch color alternates between red and green
  - Piggyback sender epoch color on message
  - If piggybacked color is not equal to receiver epoch color:
    - Receiver is logging: late message
    - Receiver is not logging: early message

Implementation details

- Out-of-band messages
  - Whenever application program does a send or receive, our thin layer also looks to see if any out-of-band messages have arrived
  - May cause a problem if a process does not exchange messages for a long time but this is not a serious concern in practice
- MPI features
  - non-blocking communication
  - Collective communication
- Save internal state of MPI library
- Write global checkpoint out to stable storage

Research issue

- Protocol is sufficiently complex that it is easy to make errors
- Shared-memory protocol
  - even more subtle because shared-memory programs have race conditions
- Is there a framework for proving these kinds of protocols correct?