Application-Level Fault Tolerance for MPI Programs

Keshav Pingali

The Problem

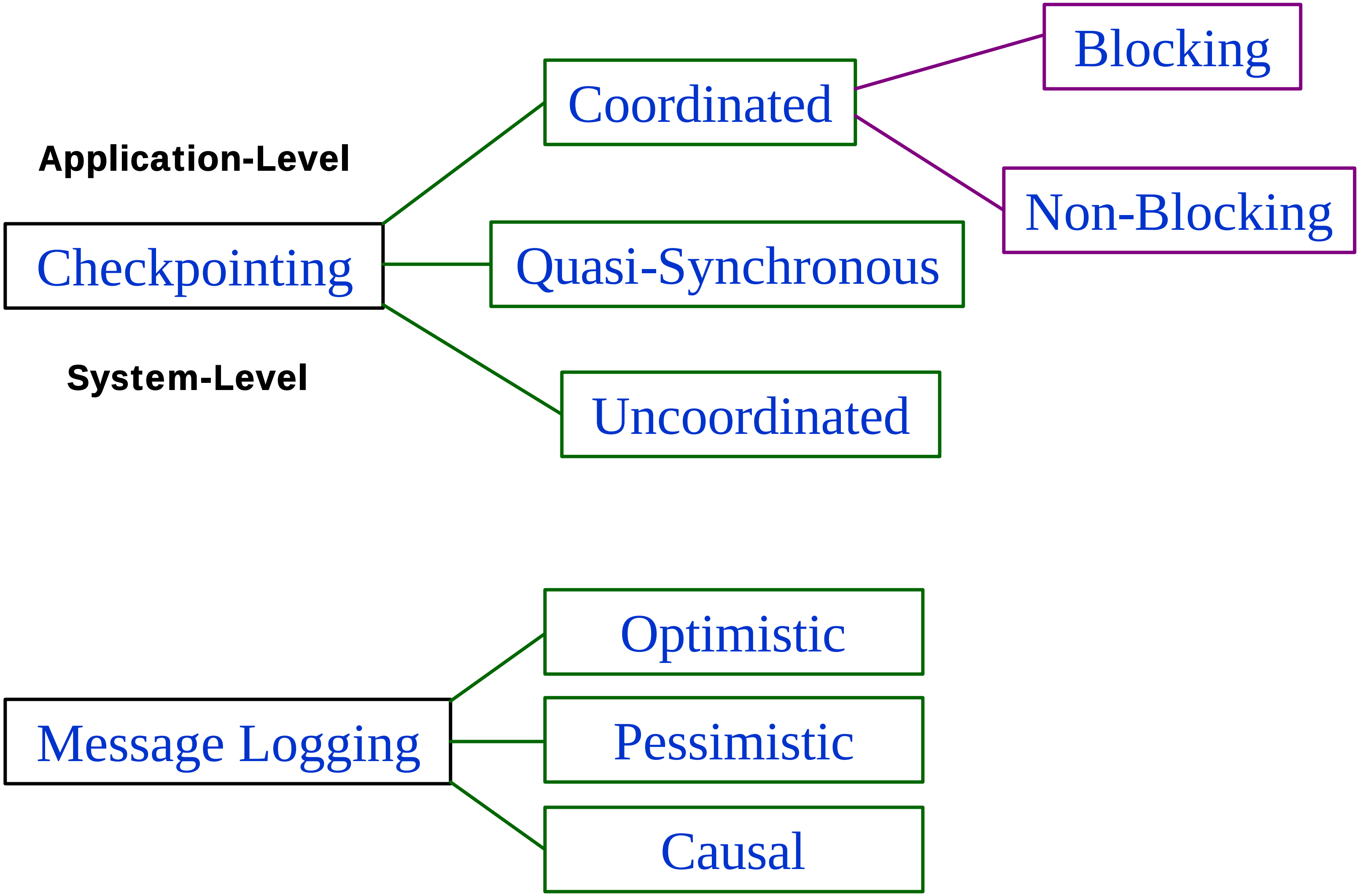
- Old picture of high-performance computing:
  - Turn-key big-iron platforms
  - Short-running codes
- Modern high-performance computing:
  - Roll-your-own platforms
    - Large clusters from commodity parts
    - Grid Computing
  - Long-running codes

- 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
- Our focus:
  - Fail-Stop Faults
  - (Semi-)automatic solution

Solution Space



Solution Space Detail

- Checkpointing *[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

Checkpointing: Two problems

- Saving the state of each process
- Coordination of checkpointing

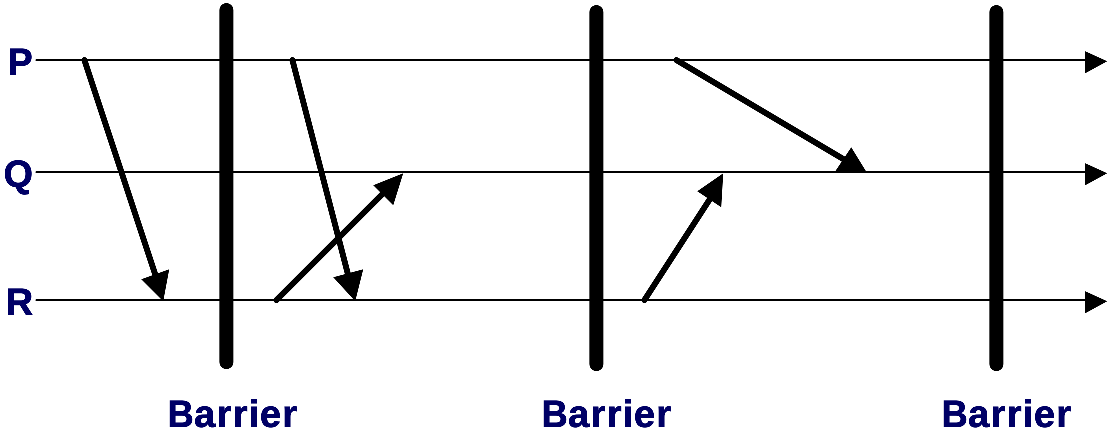
Saving process state

- System-level
  - save all bits of machine
- Application-level *[Our Choice]*
  - Programmer chooses certain points in program to save minimal state
  - Writes save/restore code
- Experience: system-level checkpointing is too inefficient for large-scale high-performance computing
  - Sandia, BlueGene

Coordinating checkpoints

- Uncoordinated
  - Dependency-tracking, time-coordinated, ...
  - Suffer from exponential rollback
- Coordinated *[Our Choice]*
  - Blocking
    - Global snapshot at a Barrier
  - Non-blocking
    - Chandy-Lamport

Blocking Co-ordinated Checkpointing

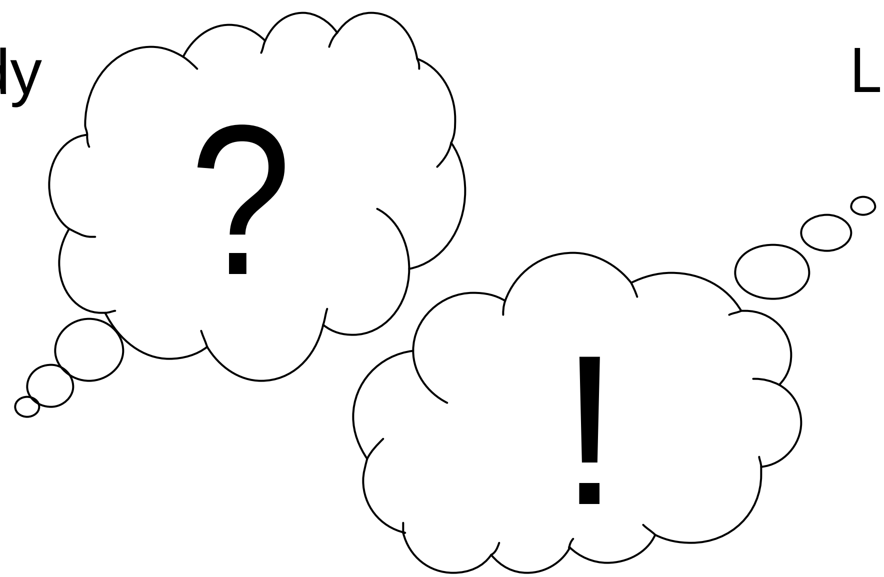


- Many programs are bulk-synchronous programs (BSP model: Valiant).
- At barrier, all processes take their checkpoints.
  - assumption: no messages are in-flight across the barrier
- Parallel program reduces to sequential state saving problem….
- but many parallel programs do not have global barriers..

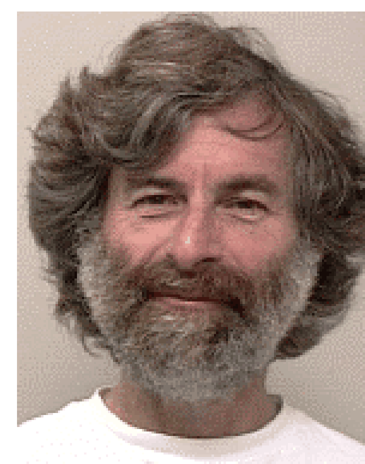
Non-blocking coordinated checkpointing

- Processes must be coordinated, but ...
- Do we really need to block ...?
- What goes wrong if saving state by processes is not co-ordinated?

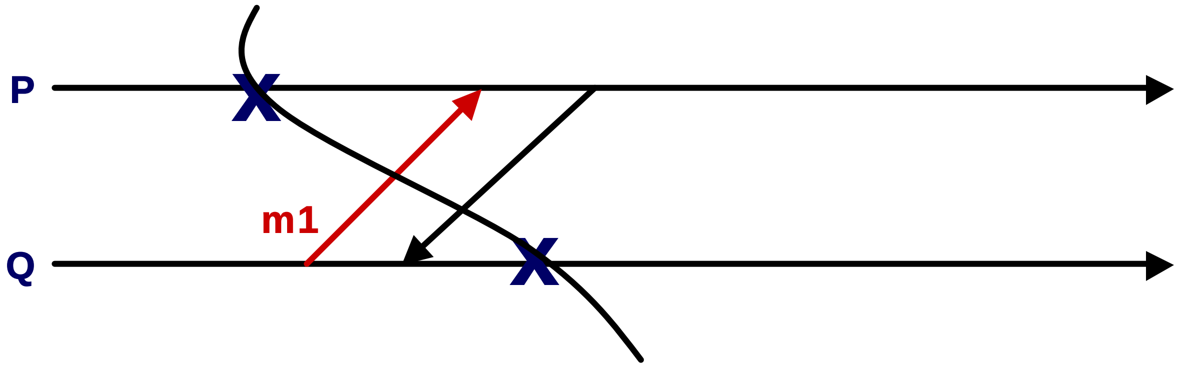
K. Mani Chandy



Leslie Lamport

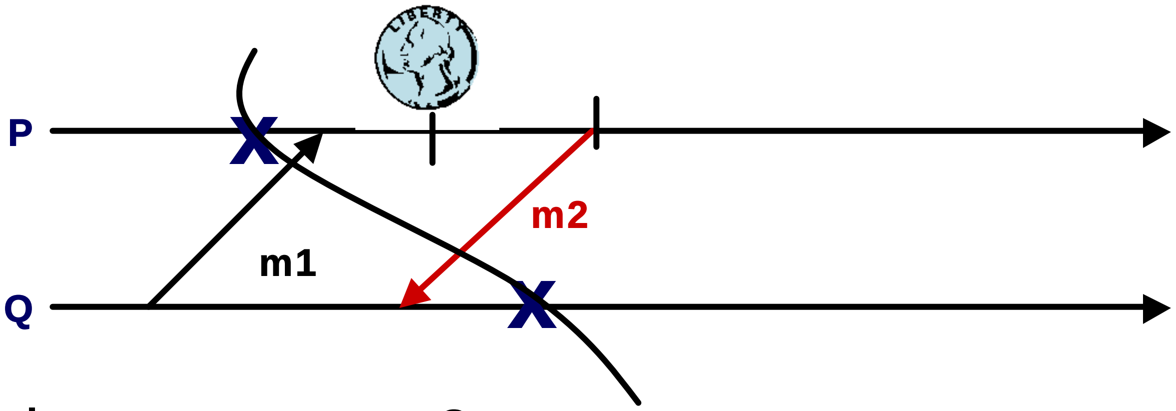


Difficulties in recovery: (I)



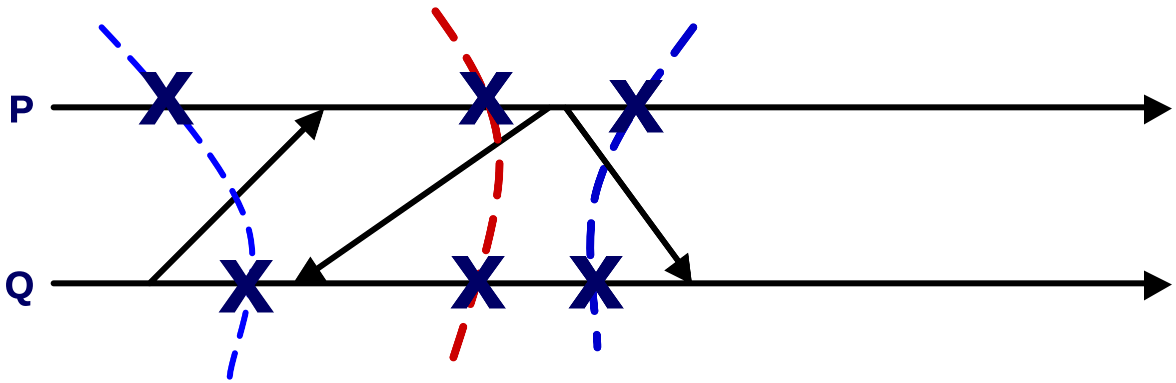
- Late message: m1
  - Q sent it before taking checkpoint
  - P receives it after taking checkpoint
- Called *in-flight* message in literature
- On recovery, how does P re-obtain message?

Difficulties in recovery: (II)



- Early message: m2
  - P sent it after taking checkpoint
  - Q receives it before taking checkpoint
- 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?
- Early messages are called inconsistent messages in literature.

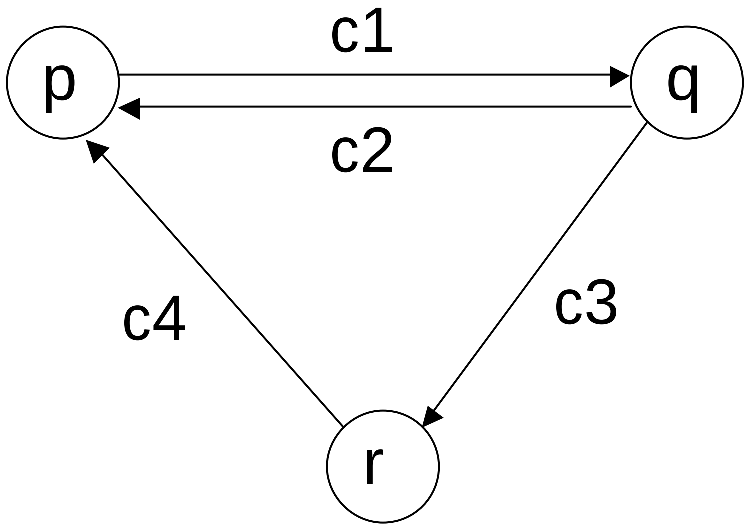
Approach in systems community



- 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 in co-ordinated checkpointing must form a consistent cut.

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. Saving process states
   – How do we avoid inconsistent messages?
2. Saving in-flight messages
3. Termination

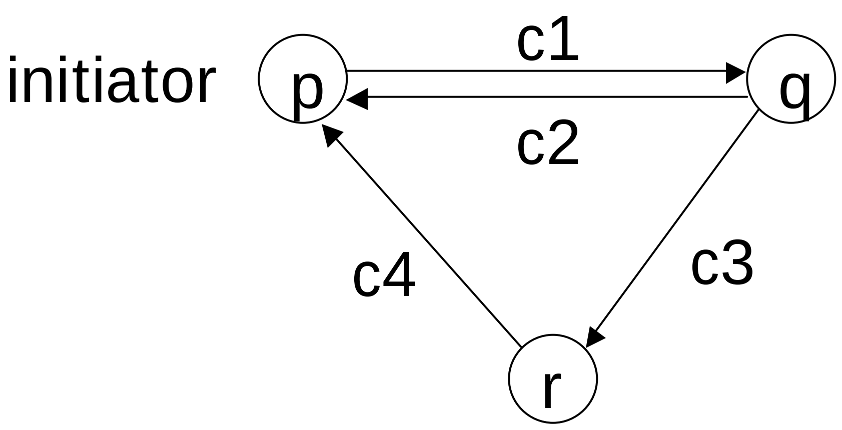
Next: Model of Distributed System

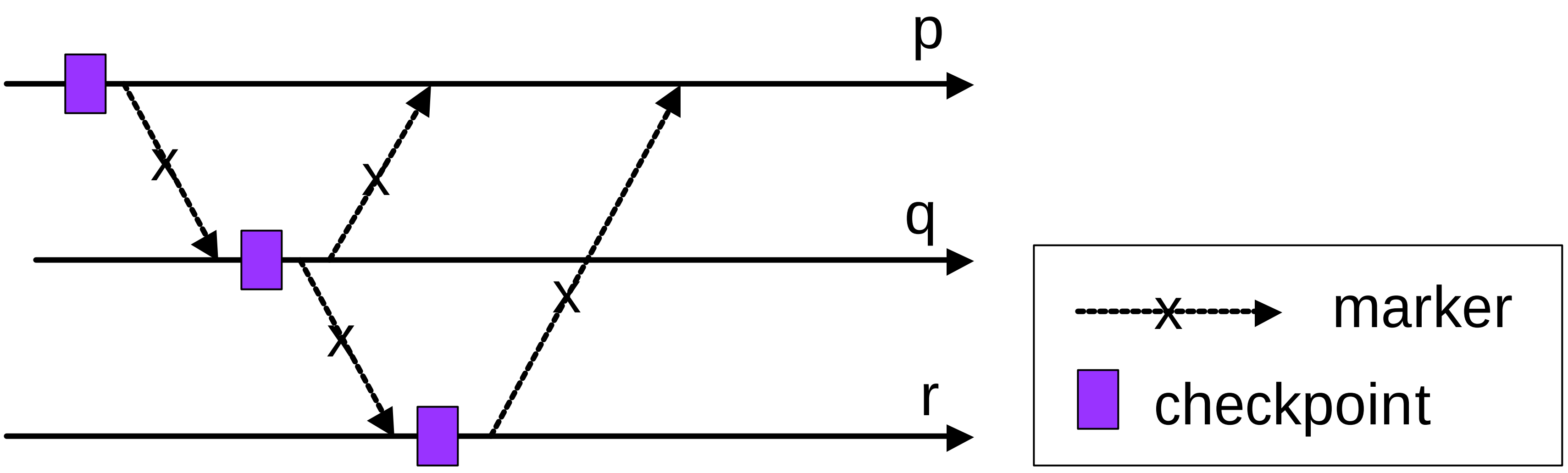
Step 1: saving process states

- Initiator:
  - Save its local state
  - Send <u>marker tokens</u> on all outgoing edges
- All other processes:
  - On receiving the first marker on any incoming edges,
    - Save state, and propagate markers on all outgoing edges
    - Resume execution.
  - Further markers will be eaten up.

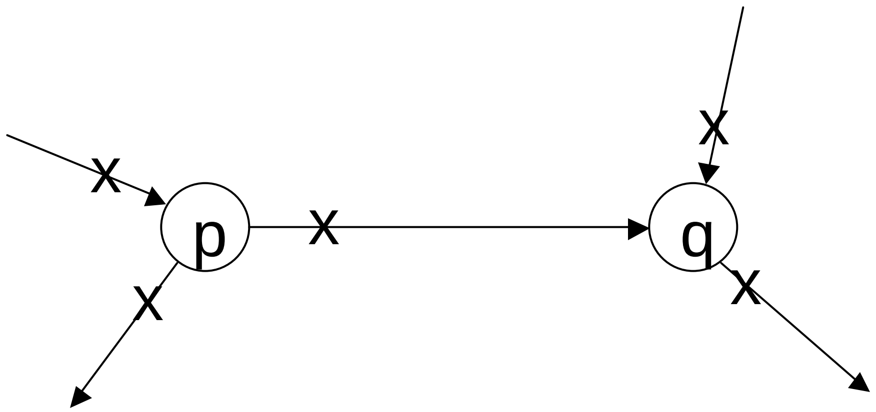
Next: Example

◆Example

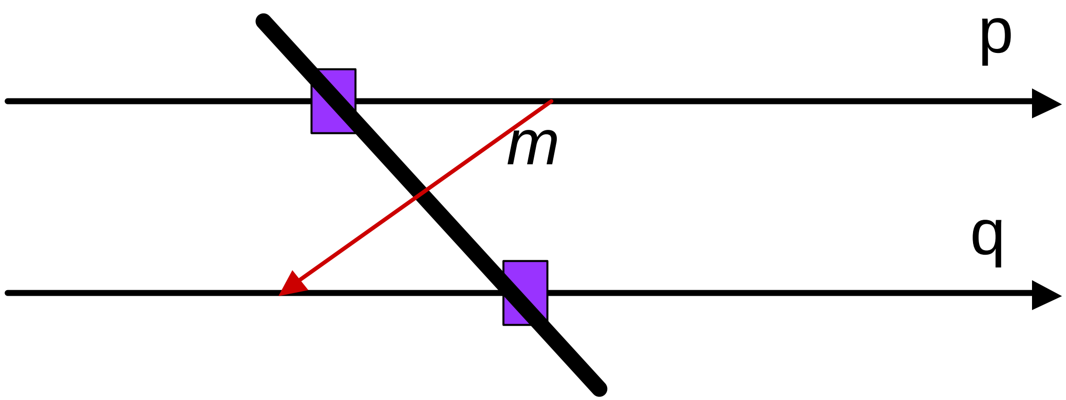




Theorem: Saved states form consistent cut

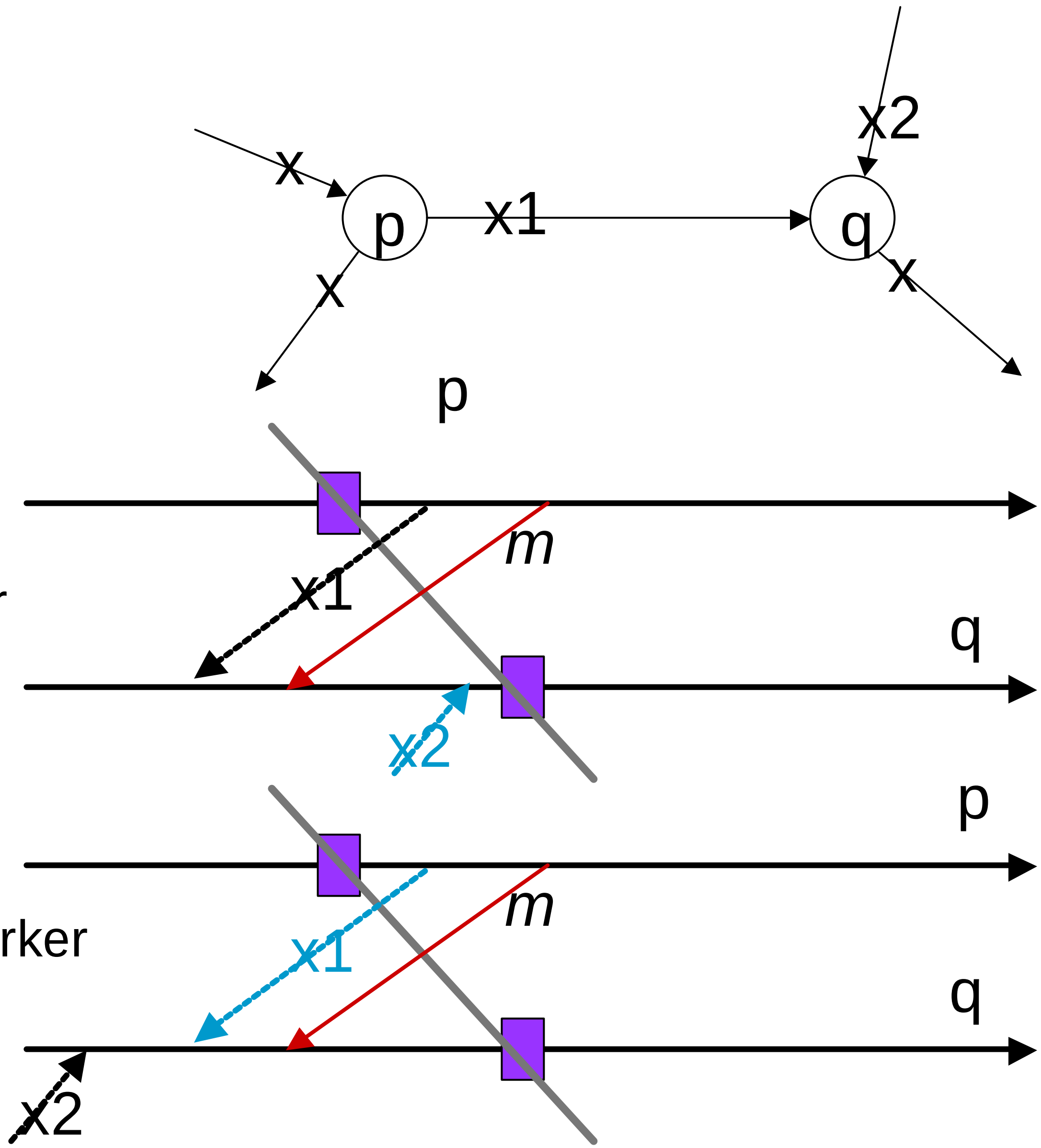


Let us assume that a message *m* exists, and it makes our cut inconsistent.



Next: Proof (cont')

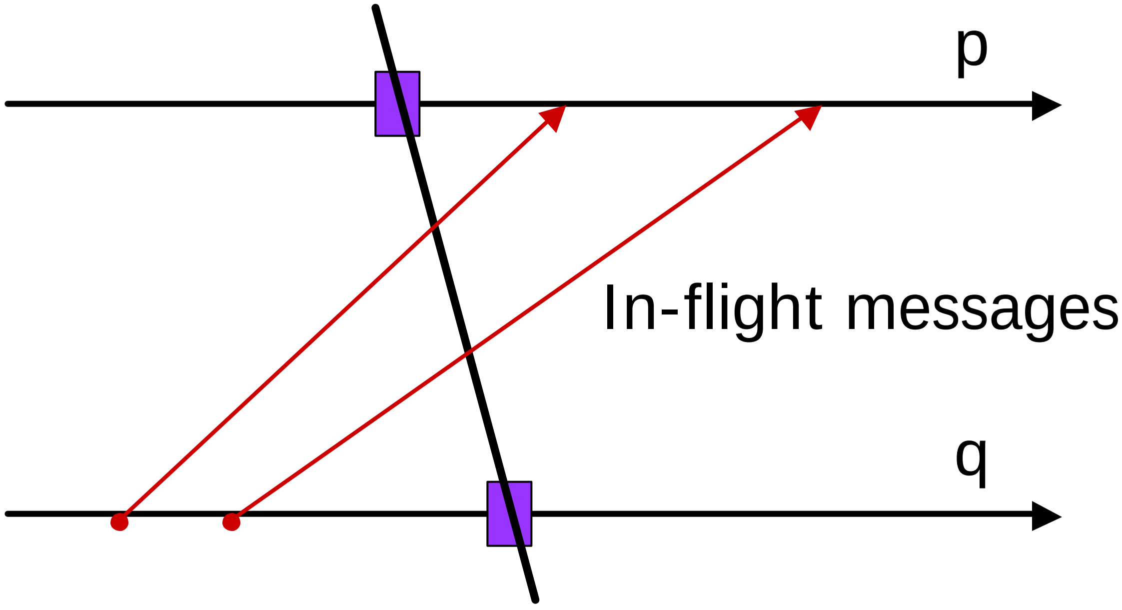
- Proof(cont')



(1) x1 <u>is</u> the 1st marker for process q

(2) x1 <u>is not</u> the 1st marker for process q

Step 2:recording in-flight messages



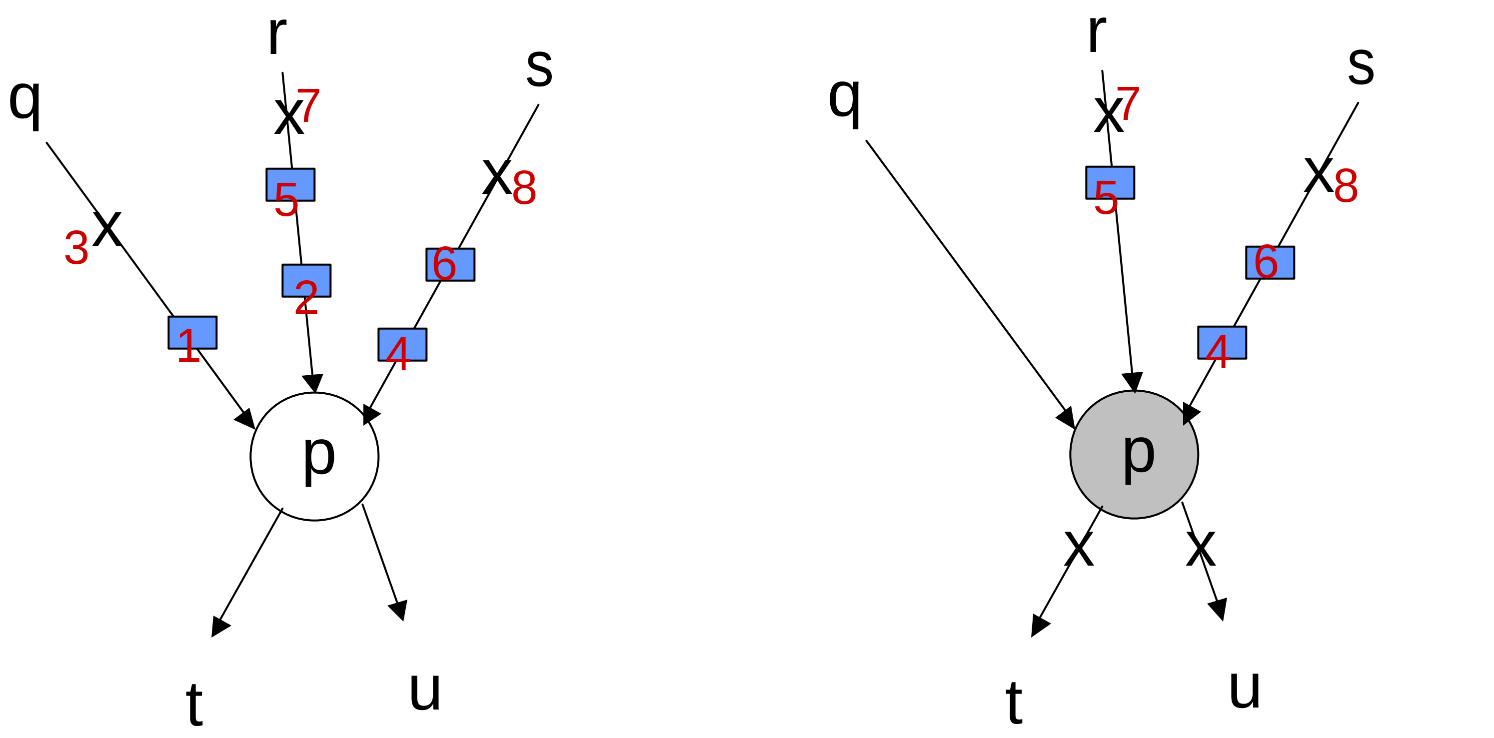
- Sent along the channel before the sender’s chkpnt
- Received along the channel after the receiver’s chkpnt

Next: Example

Example

(1) p is receiving messages

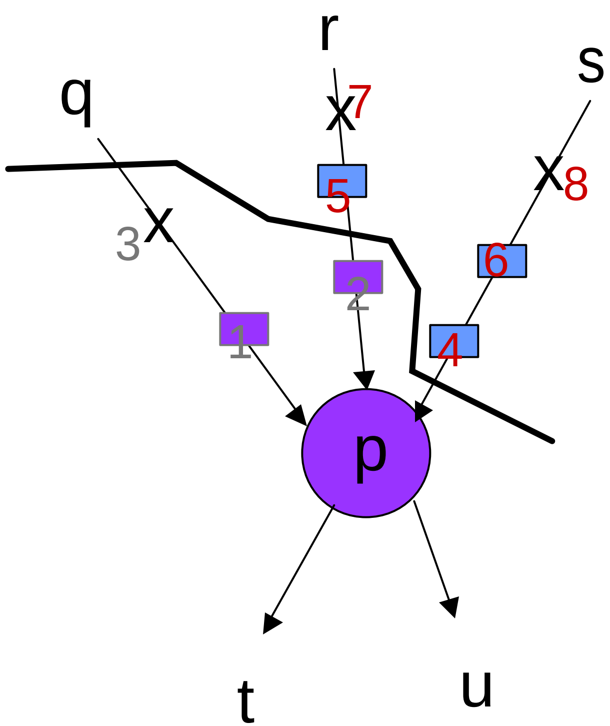
(2) p has just saved its state

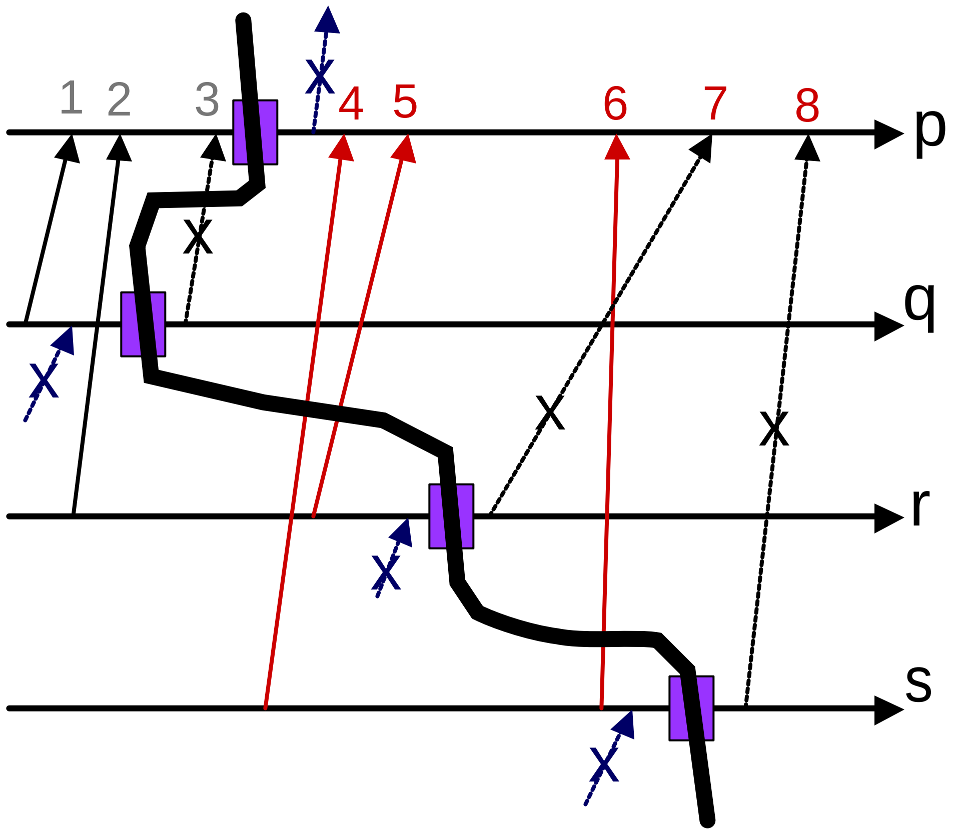


Next: Example (cont')

Example(cont')

p's chkpnt triggered by a marker from q





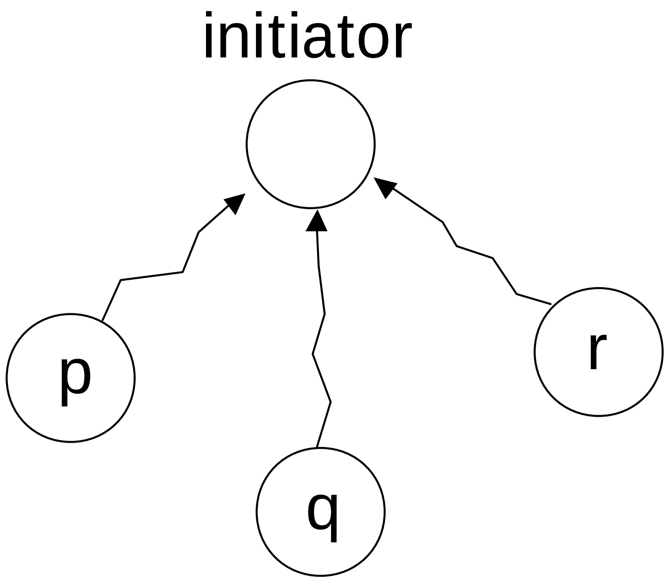
Next: Algorithm (revised)

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 each 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 each 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

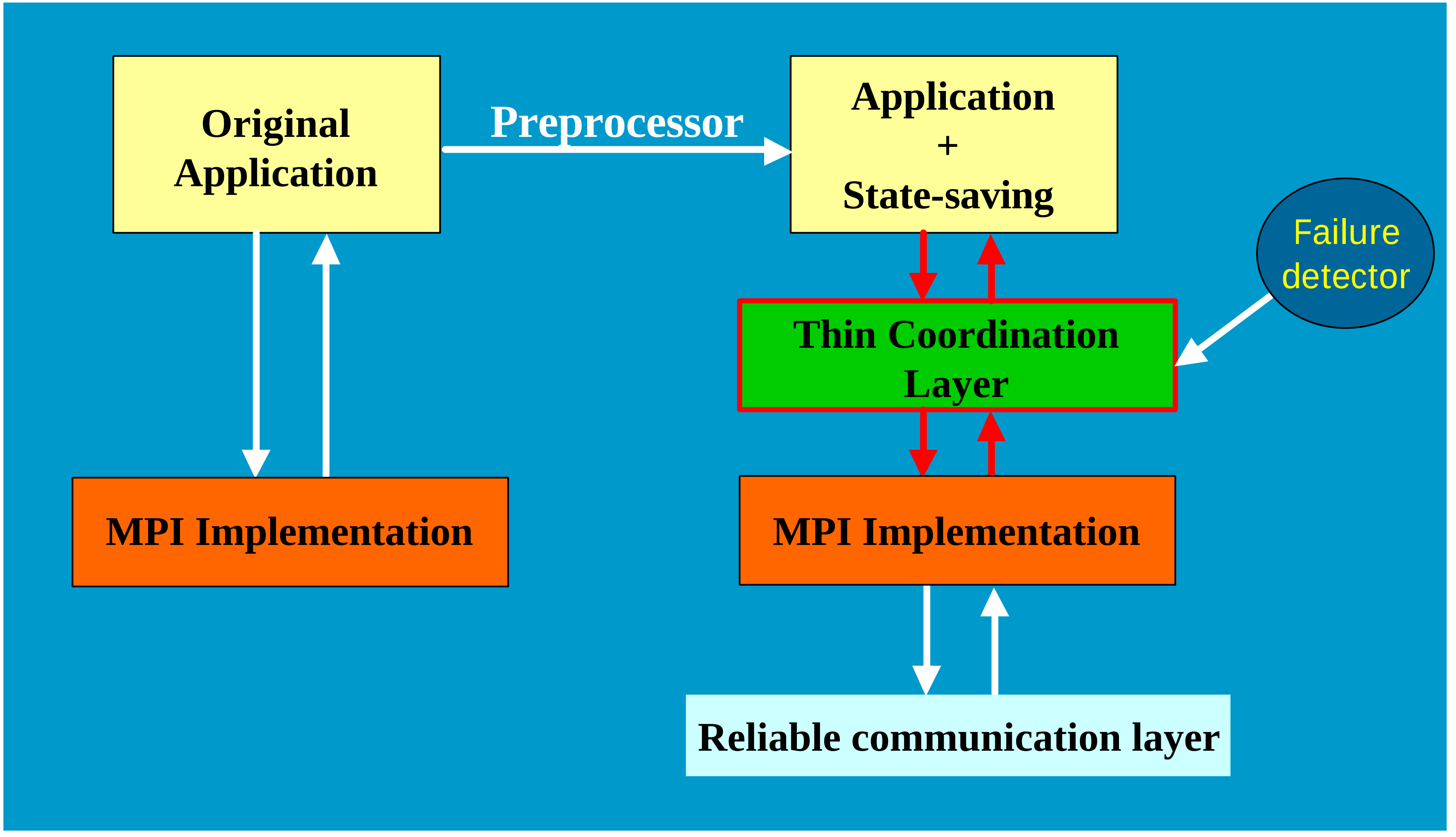


- direct channel to the initiator?
- spanning tree?

Comments on C-L protocol

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

Our approach:System Architecture



Automated Sequential Application-Level Checkpointing

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

Saving Application State

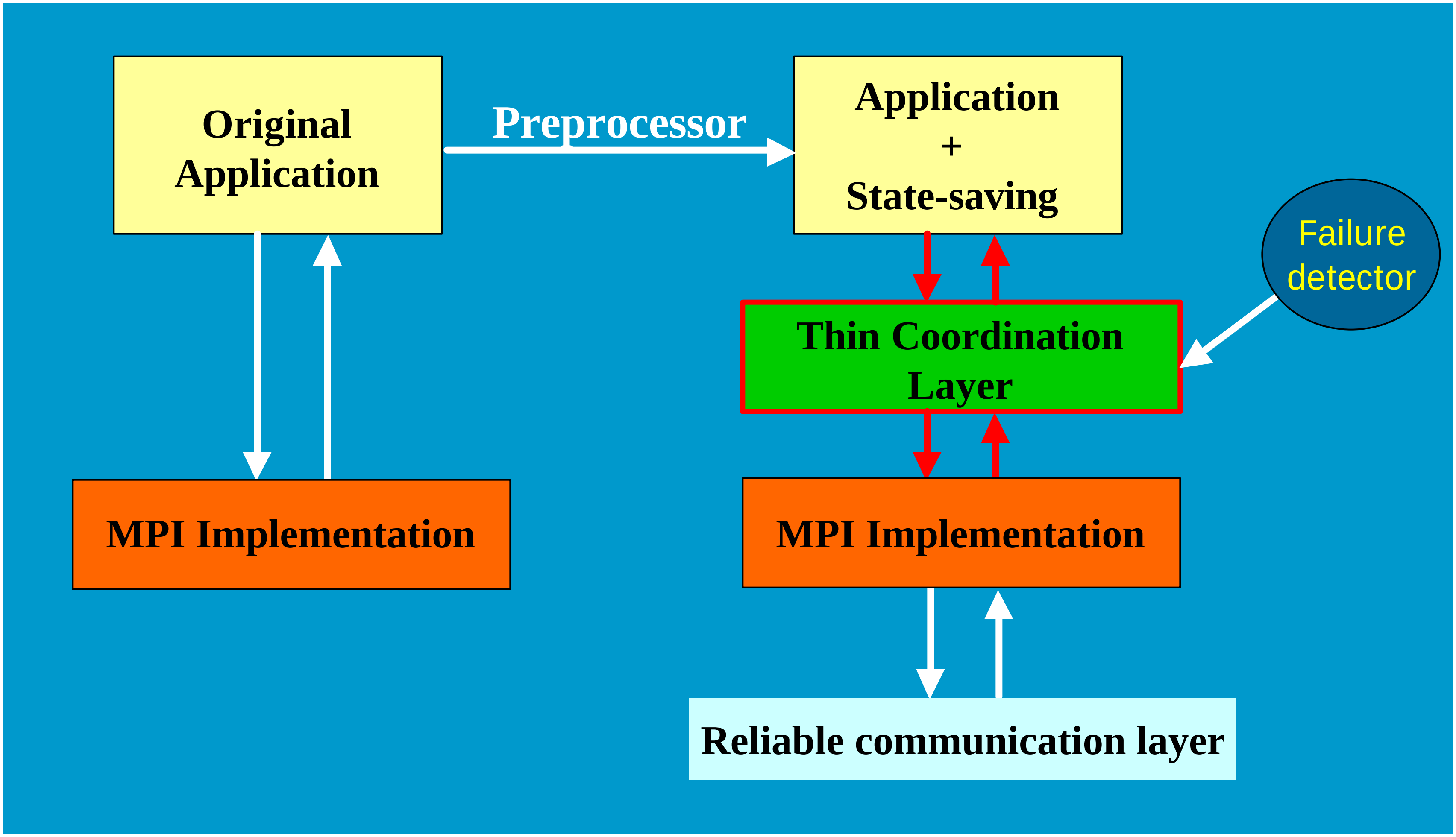
- Must save:
  - Heap – we provide special malloc that tracks the memory it allocates
  - Globals – preprocessor knows the globals; inserts statements to explicitly save them
  - Call Stack, Locals and Program Counter - maintain a separate stack which records all functions that got called and the local vars inside them.
- Similar to work done with PORCH (MIT)

Reducing saved state: Dan Marques

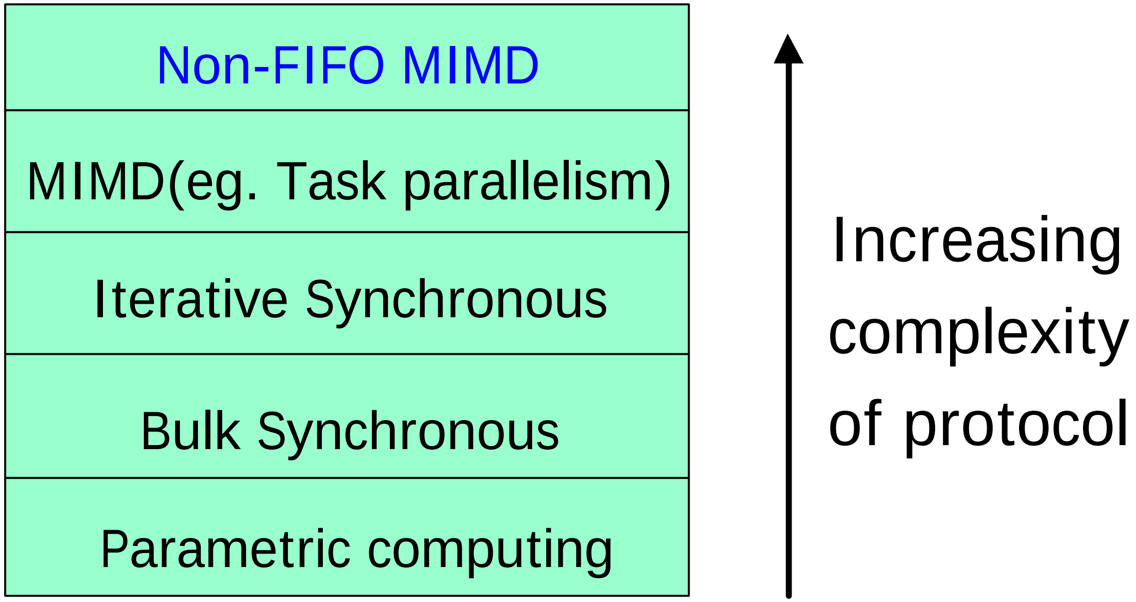
- Statically determine spots in the code with the least amount of state
- Determine live data at the time of a checkpoint
- Incremental state-saving
- Recomputation vs saving state
  - ex: Protein folding, $A \cdot B = C$
- Prior work: CATCH (Illinois).

System Architecture

Distributed Checkpointing



Distributed Checkpointing



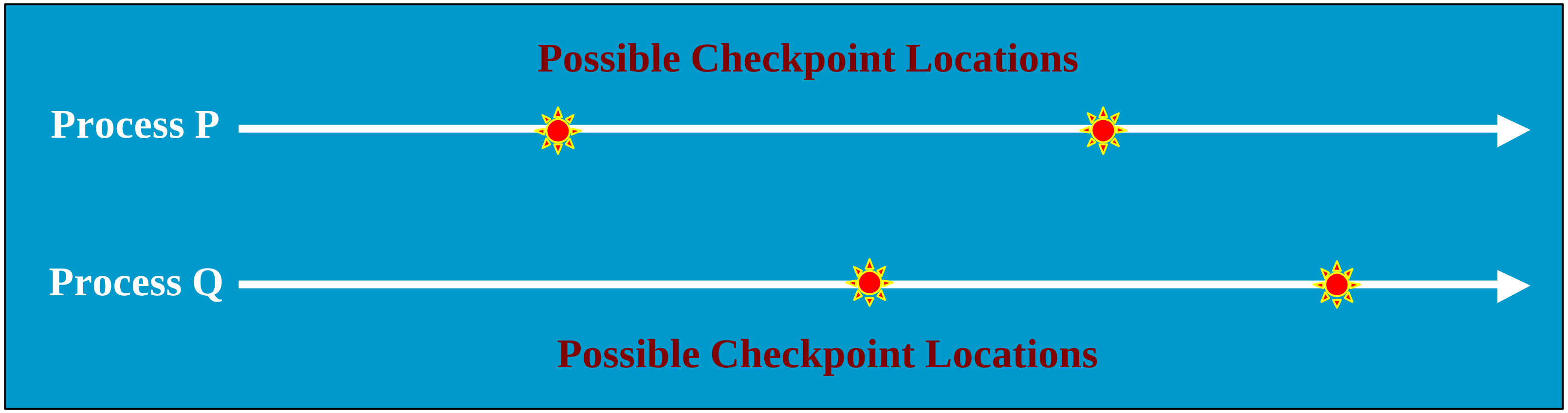
- Programs of differing communication complexity require protocols of different complexity.

Coordination protocol

- Many protocols in distributed systems literature
  - Chandy-Lamport, Time-coordinated,...
- Existing solutions
  - not applicable to application-level checkpointing

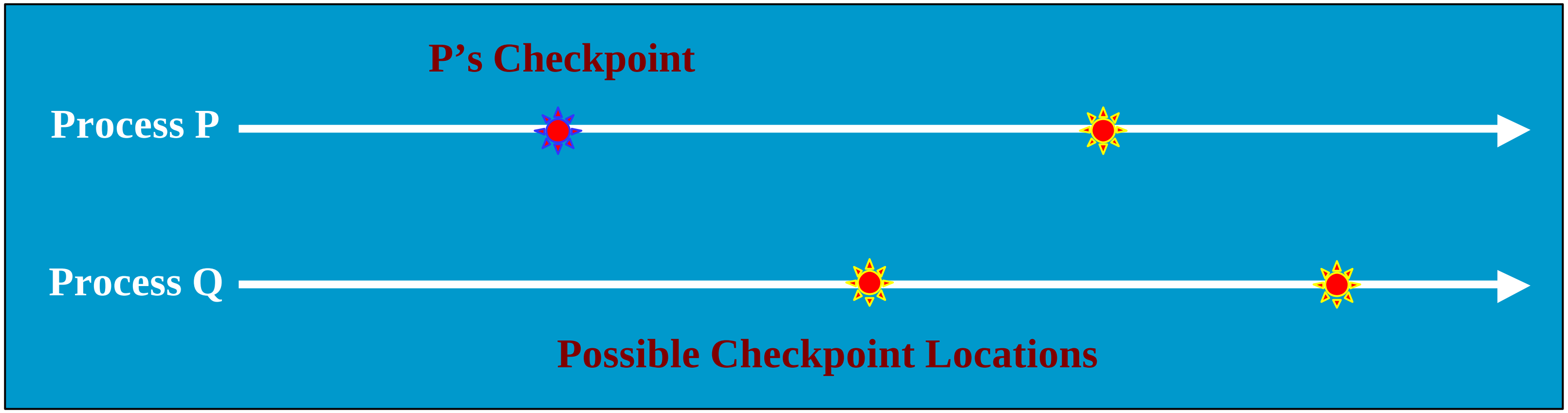
App-level difficulties

- System-level checkpoints can be taken anywhere
- Application-level checkpoints can only be taken at certain places.



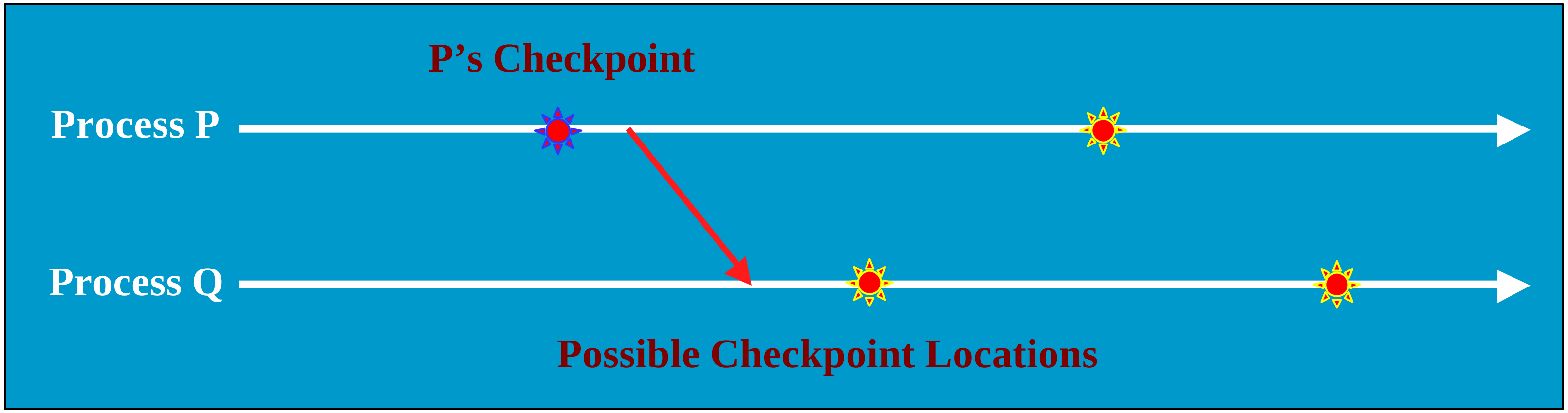
App-level difficulties

- Let P take a checkpoint at one of the available spots.



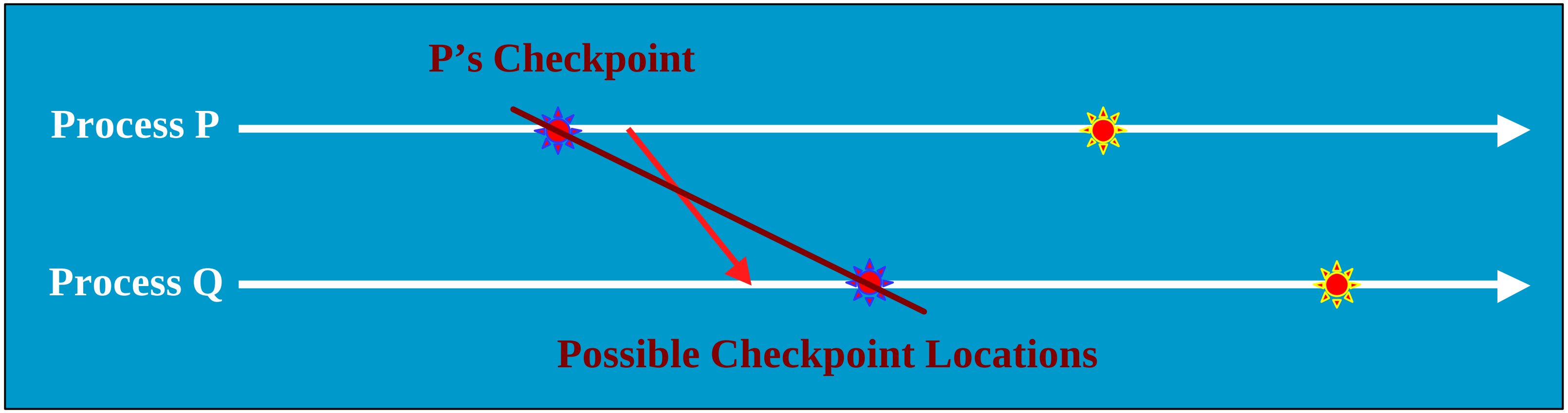
App-level difficulties

- Let P take a checkpoint at one of the available spots.
- After checkpointing, P sends a message to Q.

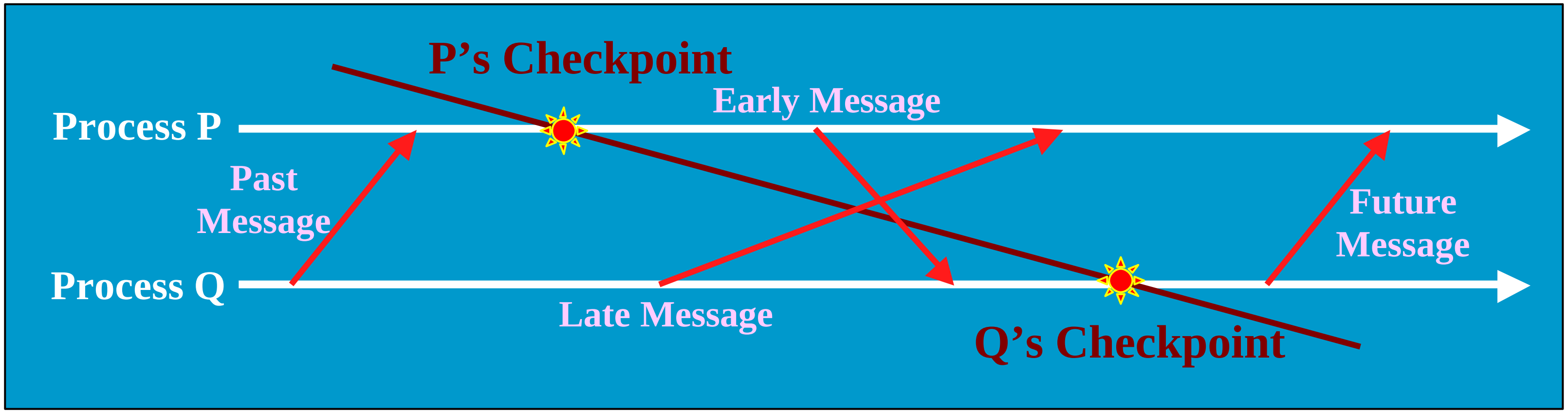


App-level difficulties

- The next possible checkpoint on Q is too late.
- The only possible recovery lines make this an inconsistent message.

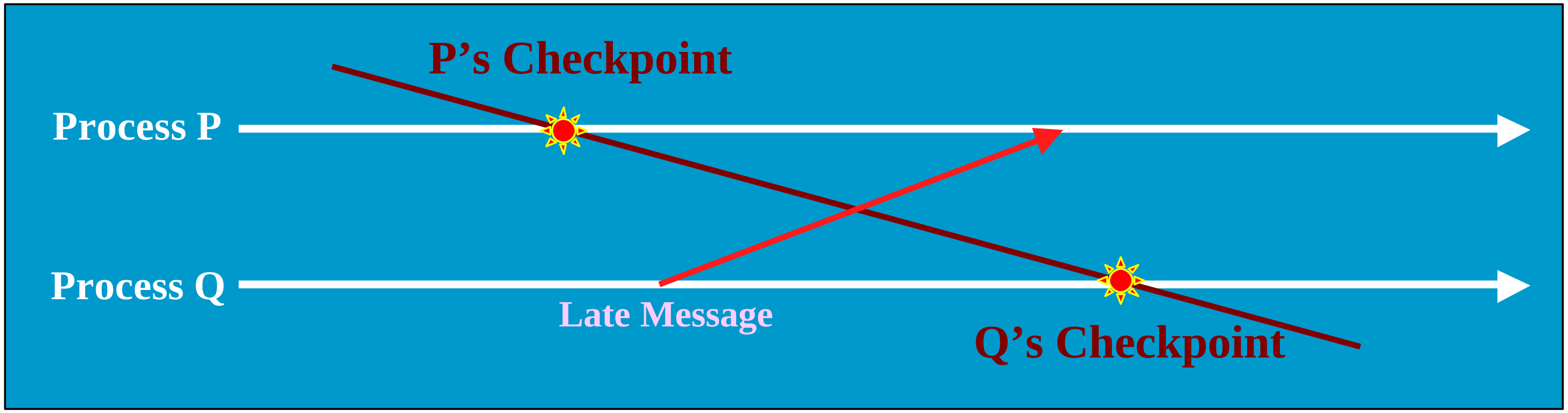


Possible Types of Messages



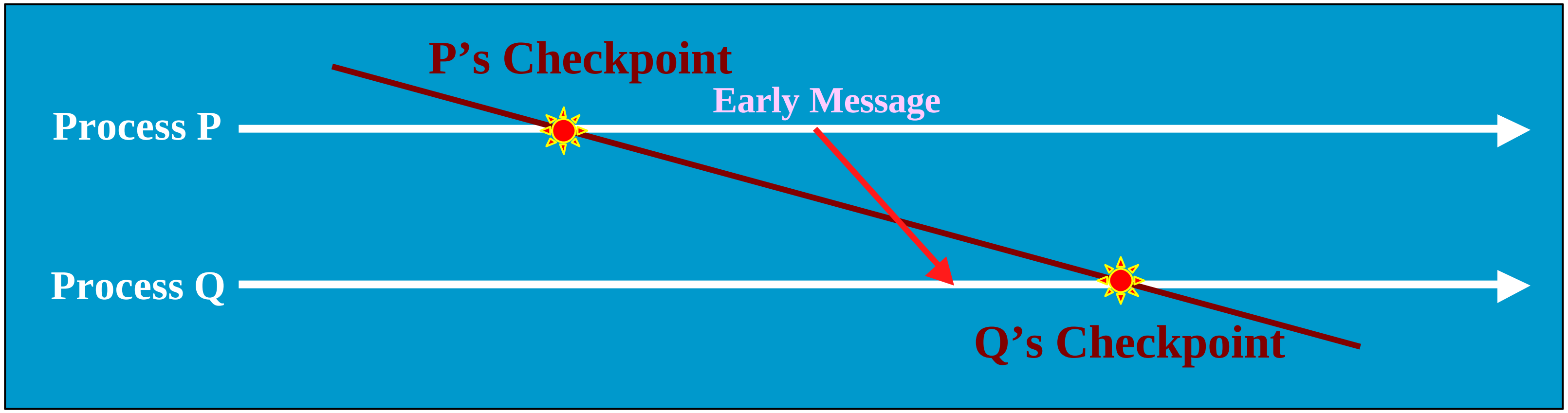
- On Recovery:
  - Past message will be left alone.
  - Early message will be re-received but not resent.
  - Late message will be resent but not re-received.
  - Future message will be reexecuted.

Late Messages



- To recover we must either:
  – Record message at sender and resend it on recovery.
  – Record message at receiver and re-read it from the log on recovery. *[Our choice]*

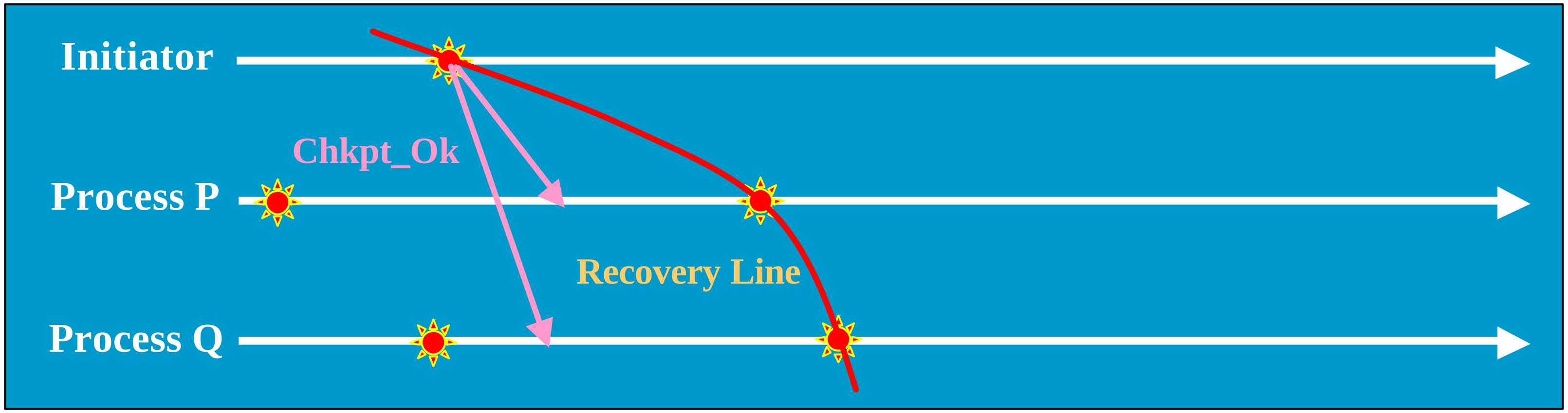
Early Messages



- To recover we must either:
  - Reissue the receive, allow application to resend.
  - Suppress resend on recovery. *[Our choice]*
- Must ensure the application generates the same message on recovery.

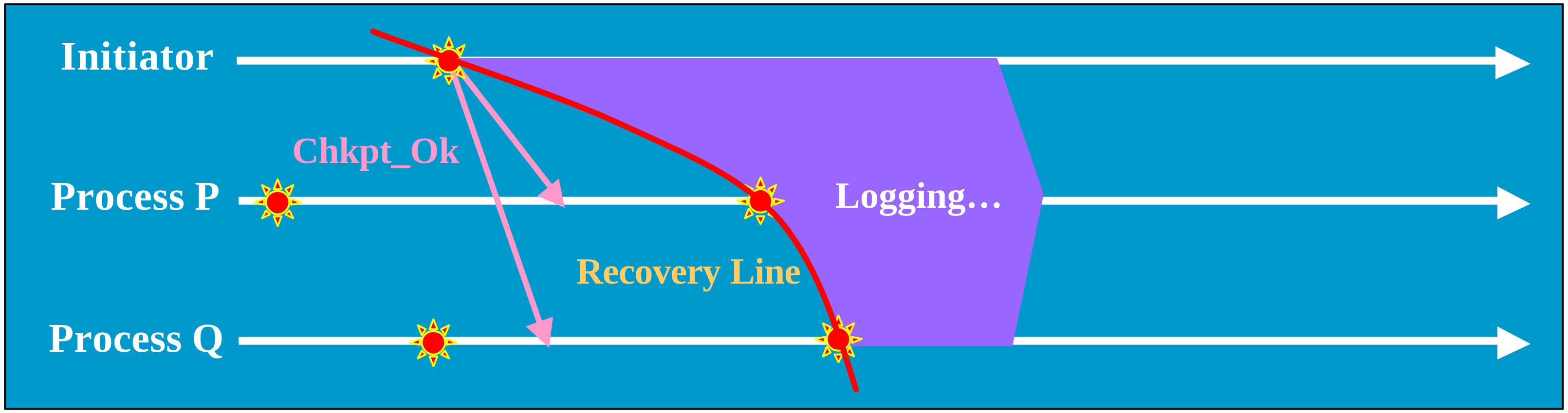
The Protocol

High-level view of our protocol: (I)



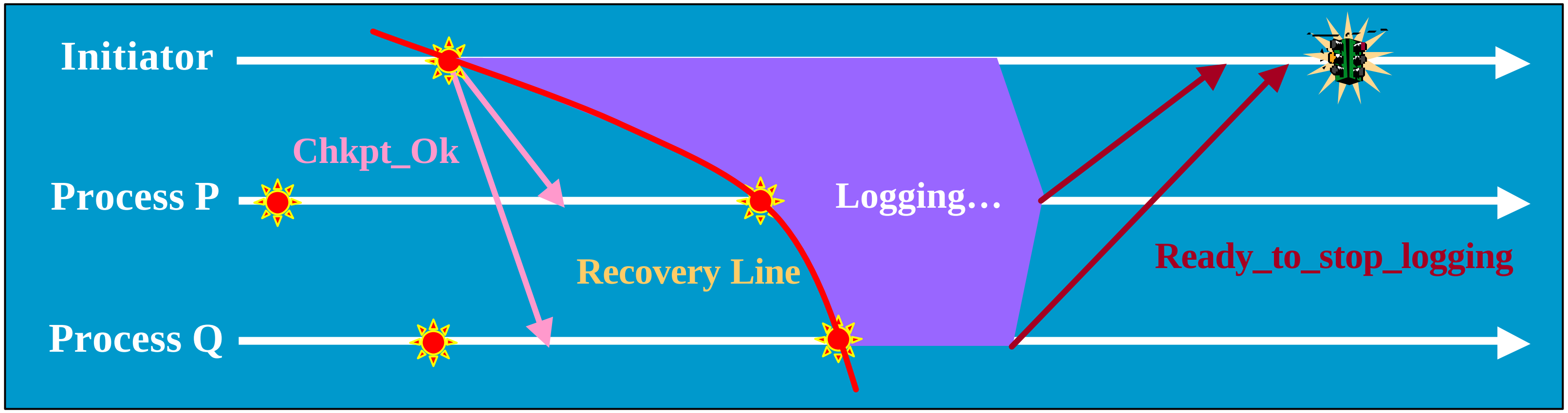
- The initiator takes a checkpoint and sends everyone a Chkpt_Ok message.
- After a process receives this message, it takes a checkpoint at the next available spot

High-level view of our protocol: (II)



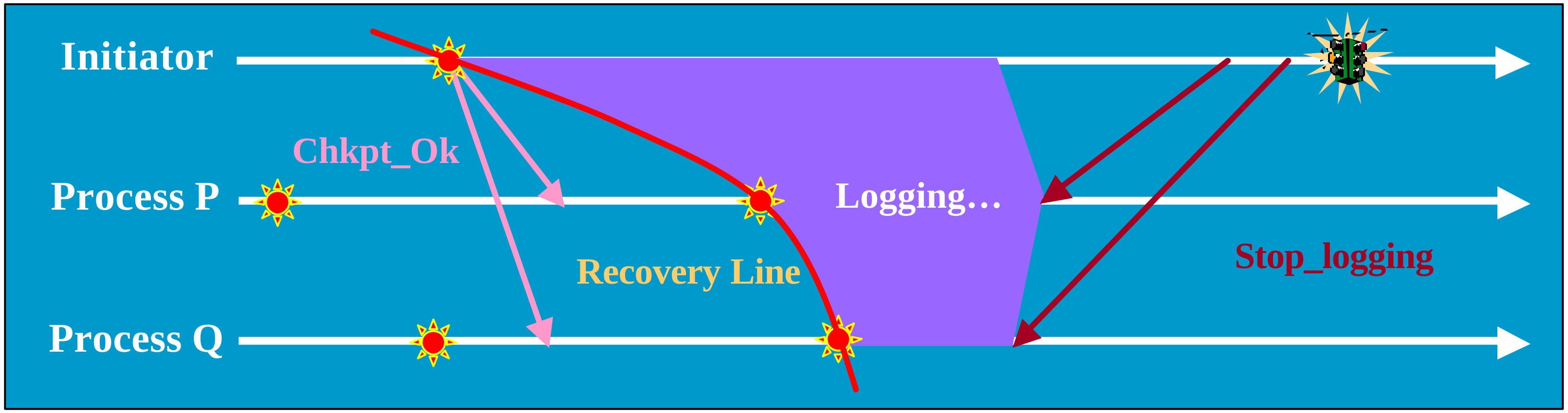
- After taking a checkpoint each process keeps a log.
- This log records message data and non-deterministic events.

High-level view of our protocol: (III)



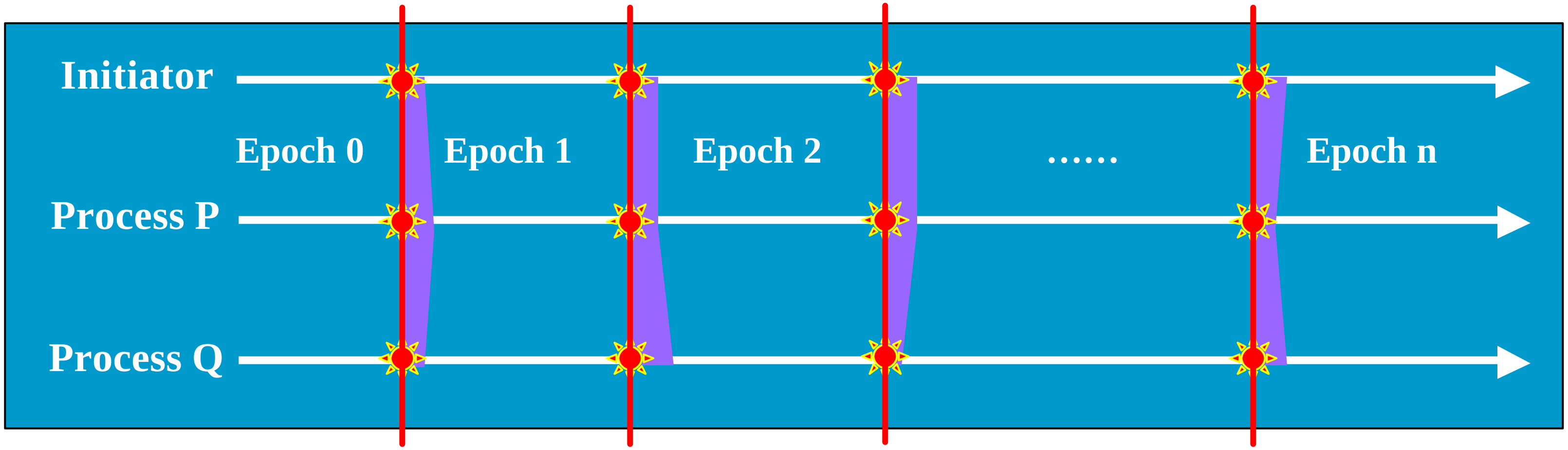
- When a process is ready to stop logging, it sends the Initiator a Ready-to_stop_logging message.
- When the Initiator receives these messages from all processors, it knows all processes have crossed the recovery line.

High-level view of our protocol: (IV)



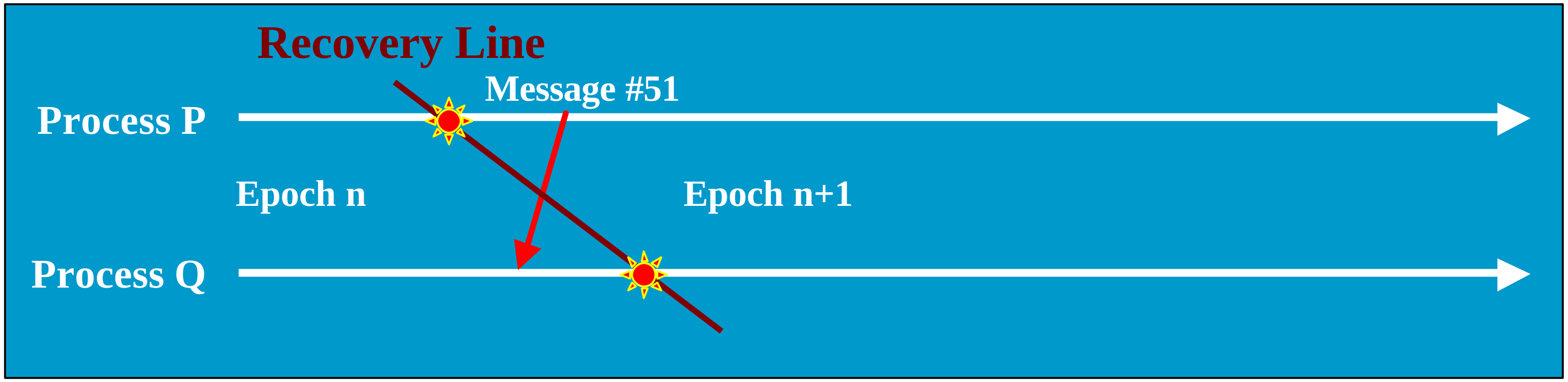
- When initiator gets Ready-to_stop_logging message fro mall processes, it sends Stop_logging messages to all processes.
- When process receives message, it stops logging and saves log on disk.

The Global View



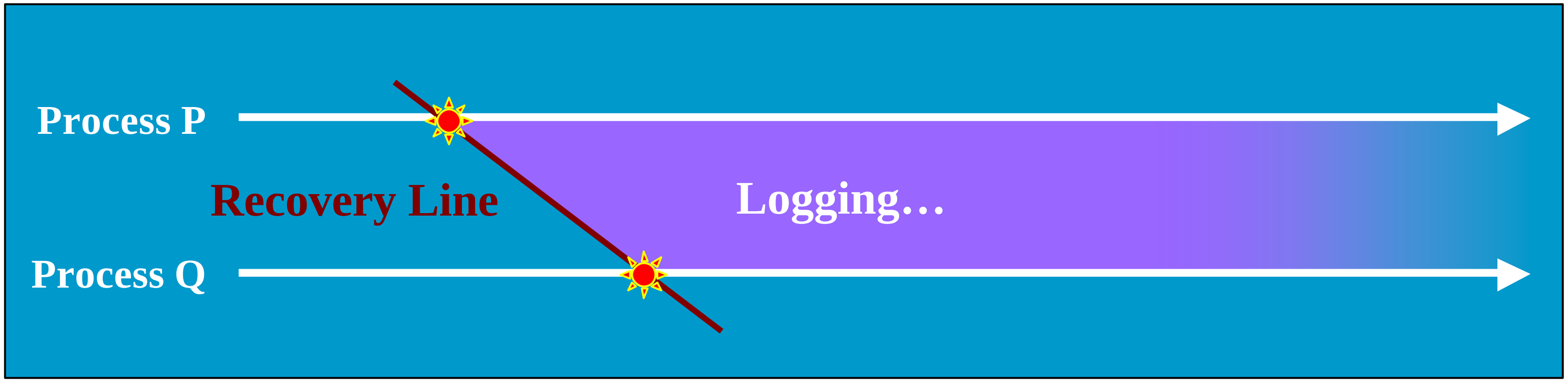
- A program’s execution is divided into a series of disjoint epochs
- Epochs are separated by recovery lines
- A failure in Epoch n means all processes roll back to the prior recovery line

Mechanism: Control Information



- Attach to each outgoing message
  - A unique message ID
  - The number of the current Epoch
  - Bit that says whether we're currently logging
- In practice: 2 bits are sufficient
- Use this to determine whether message is late/early etc.

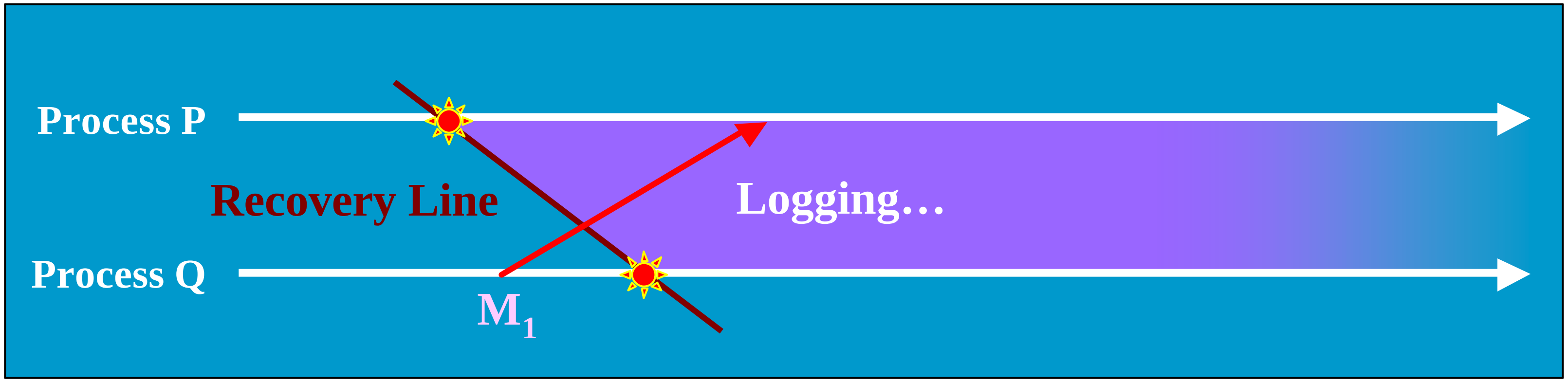
Mechanism: The Log



- Keep a log after taking a checkpoint
- During Logging phase
  - Record late messages at receiver
  - Log all non-deterministic events

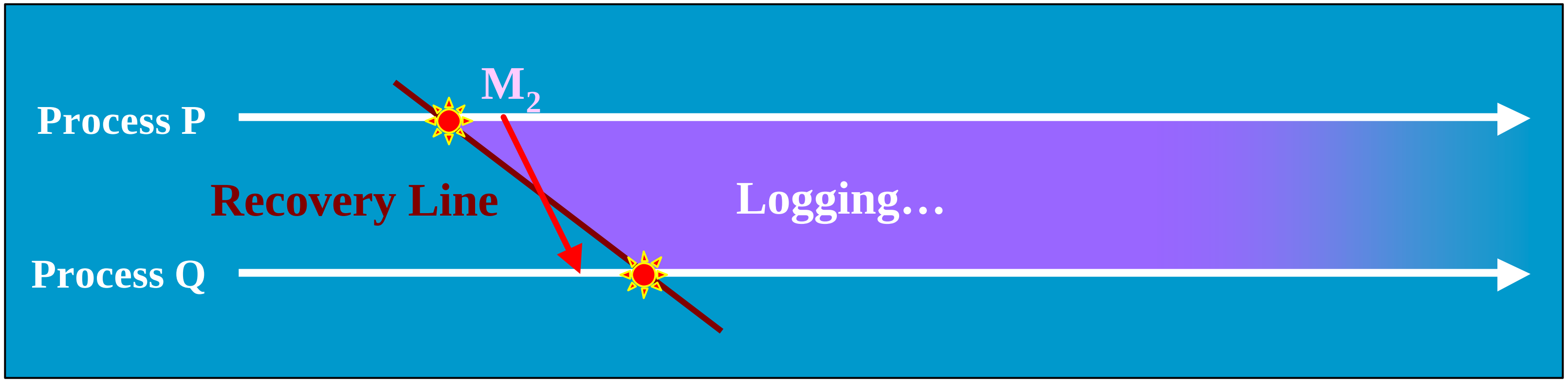
    ex: rand(), MPI_Test(), MPI_Recv(ANY_SOURCE)

Handling Late Messages



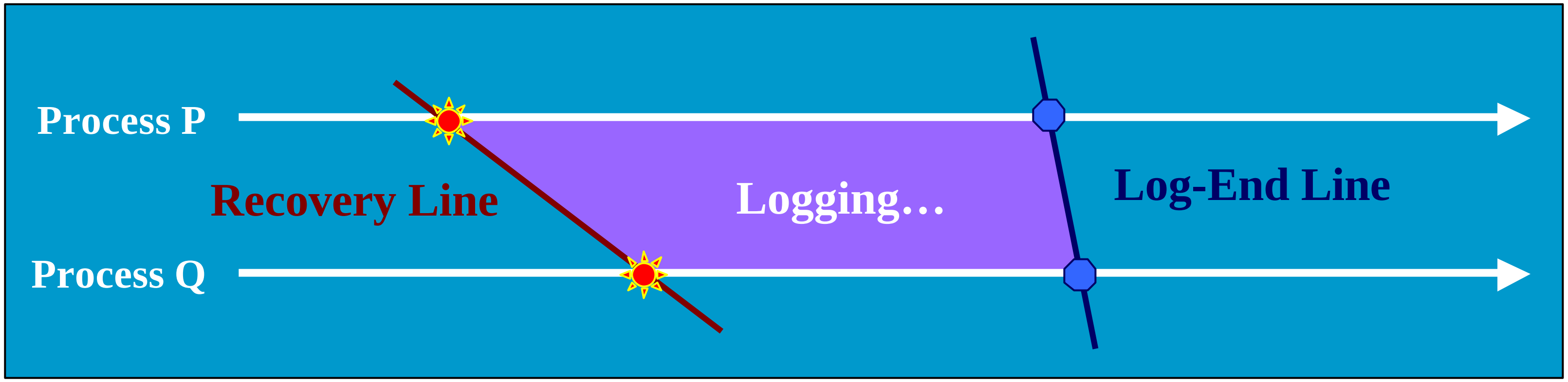
- We record its data in the log
- Replay this data for the receiver on recovery

Handling Early Messages



- Early messages sent before logging stops
  - On recovery they're recreated identically
- The receiver records that this message must be suppressed and informs the sender on recovery.

Log-End Line



- Terminate log to preserve these semantics:
  - No message may cross Log-End line backwards
  - No late message may cross Log-End line
- Solution:
  - Send Ready_to_stop_logging message after receiving all late messages
  - Process stops logging when it receives Stop_log message from initiator or when it receives a message from a process that has itself stopped logging

Additional Issues

- How do we
  - Deal with non-FIFO channels? (MPI allows non-FIFO communication)
  - Write the global checkpoint out to stable storage?
  - Implement non-blocking communication?
  - Save internal state of MPI library?
  - Implement collective communication?

Collective Communication

- Single communication involving multiple processes
  - Single-Sender: one sender, multiple receivers

    ex: Bcast, Scatter
  - Single-Receiver: multiple senders, one receiver

    ex: Gather, Reduce
  - AlltoAll: every process in group sends data to every other process

    ex: AlltoAll, AllGather, AllReduce, Scan
  - Barrier: everybody waits for everybody else to reach barrier before going on.

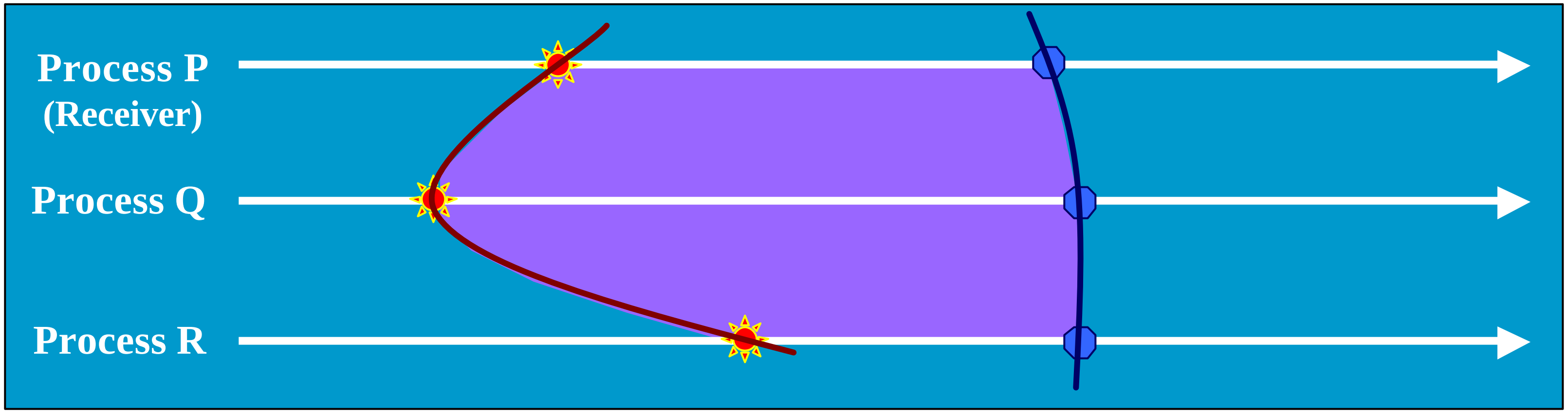
    (Only collective call with explicit synchronization guarantee)

Possible Solutions

- We have a protocol for point-to-point messages. Why not reimplement all collectives as point-to-point messages?
  - Lots of work and less efficient than native implementation.
- Checkpoint collectives directly without breaking them up.
  - May be complex but requires no reimplementation of MPI internals.

Single-Receiver Collectives

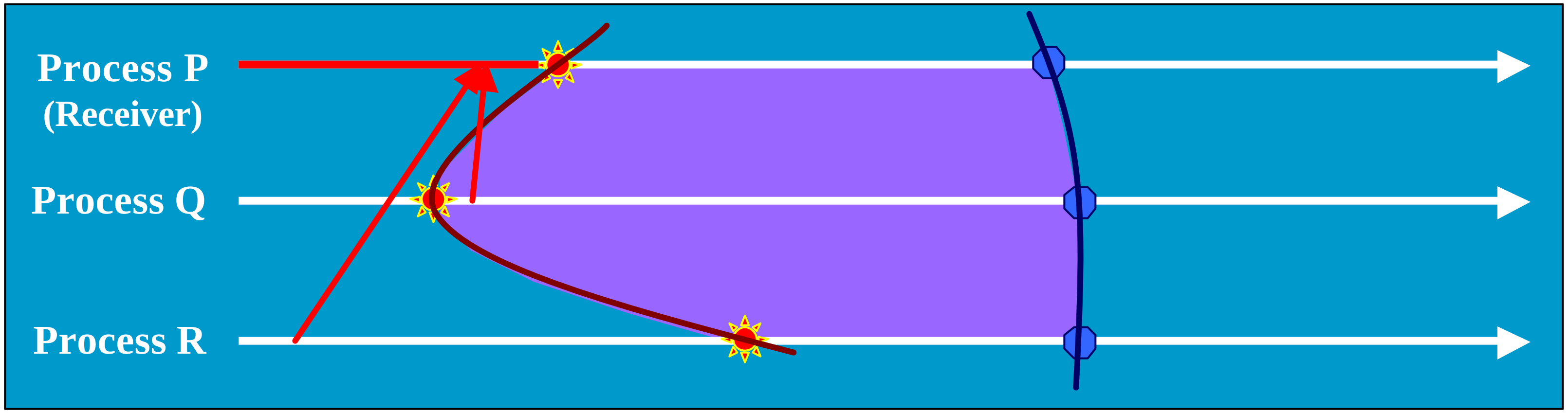
MPI_Gather(), MPI_Reduce()



- In a Single-Receiver Collective the receiver may be in one of three regions
  – Before checkpoint
  – Inside Log
  – After Log

Single-Receiver Collectives

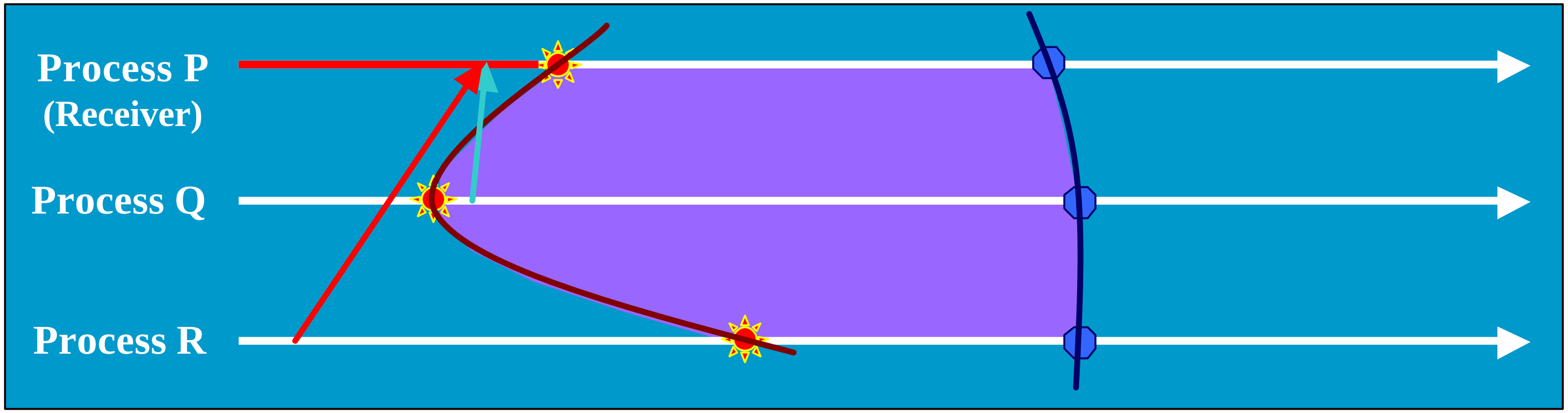
Receive is before the checkpoint



- If the Receive is before the Recovery Line sends could only have occurred:
  - Behind the Recovery Line
  - Inside the Log

Single-Receiver Collectives

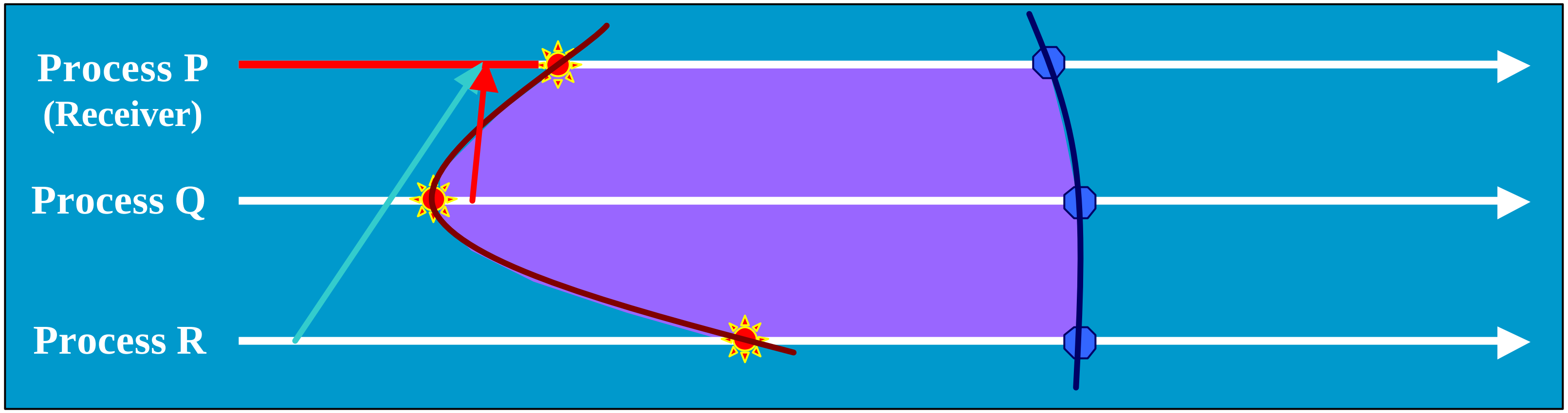
Receive is before the checkpoint



- The send from behind the recovery line will not be reexecuted.
- We should leave it alone if possible.

Single-Receiver Collectives

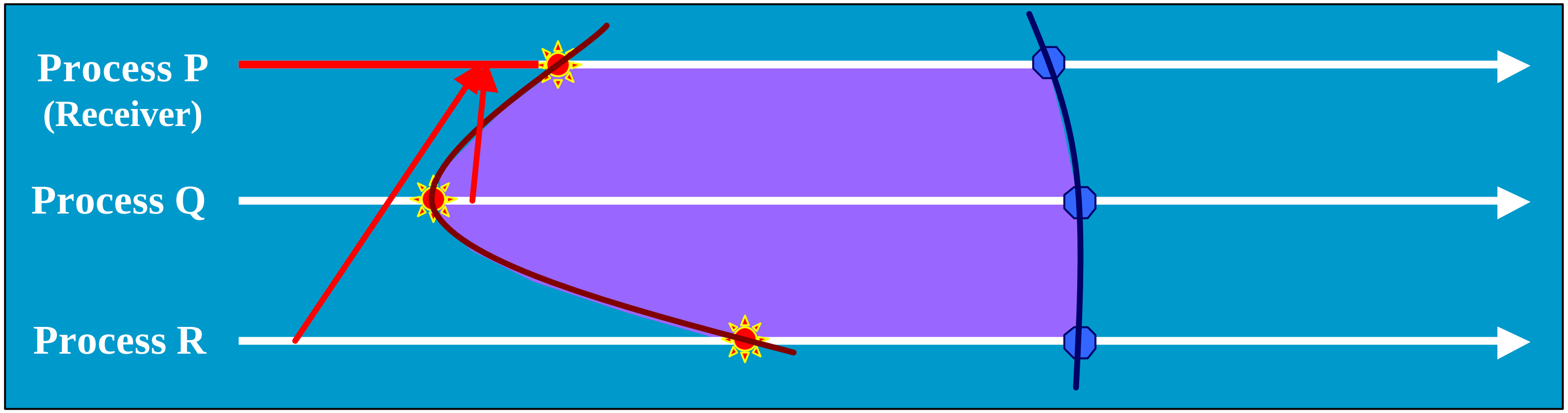
Receive is before the checkpoint



- The send from inside the log will be reexecuted.
- We already got its data and it will be regenerated with the same data.
- Thus, we should suppress it.

Single-Receiver Collectives

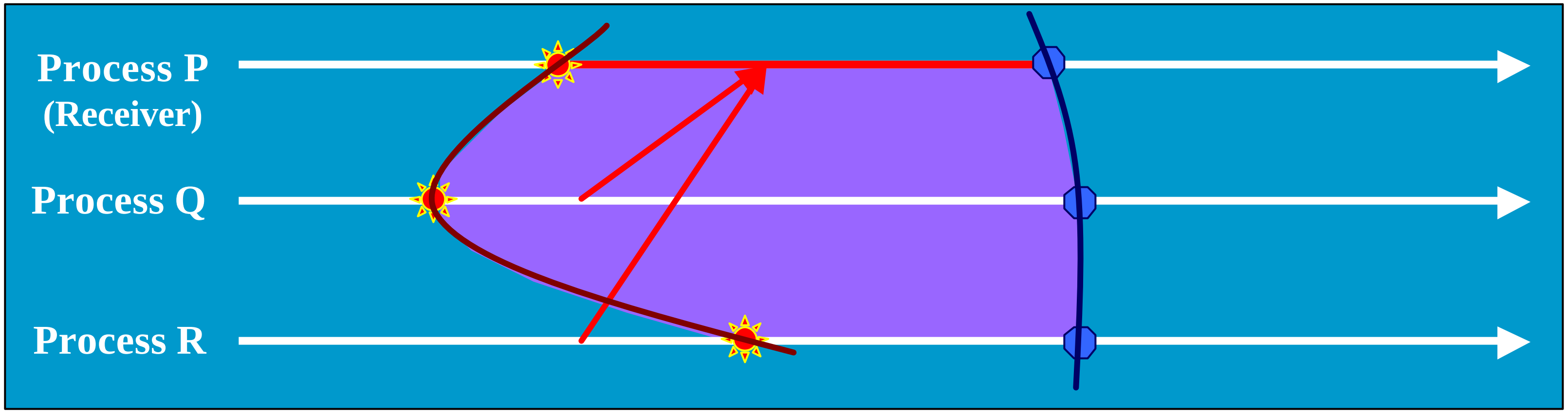
Receive is before the checkpoint



- Therefore, since neither Q or R will resend, we don't need to re-receive!

Single-Receiver Collectives

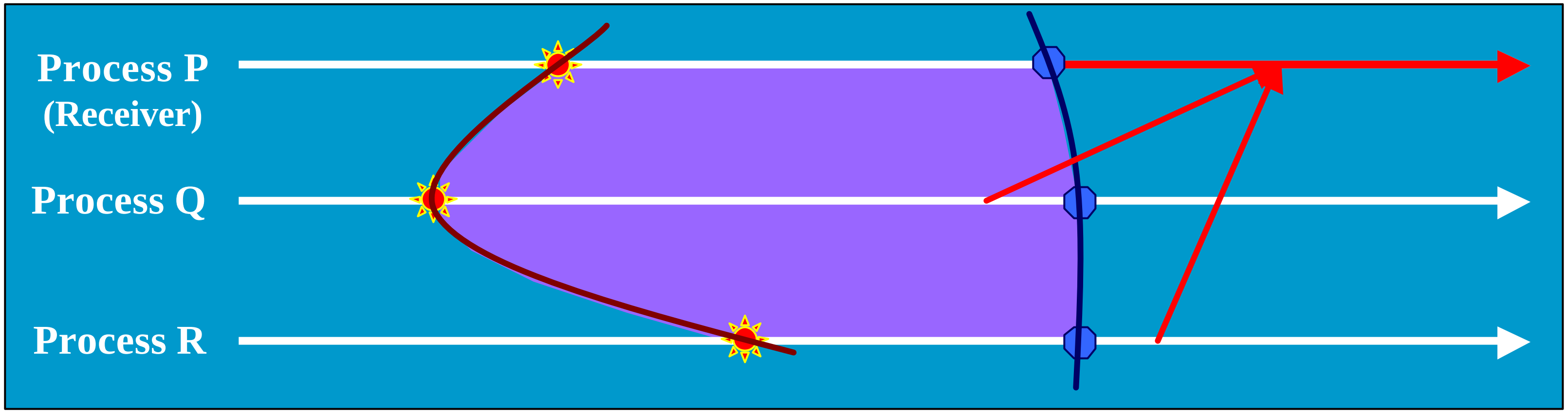
Receive is inside the log



- If the Receive is inside the log sends could only have occurred:
  - Behind the Recovery Line
  - Inside the Log
- We will log/suppress these collectives.

Single-Receiver Collectives

Receive is after the log



- If the Receive is after the log sends could only have occurred:
  - Inside the Log
  - After the Log
- We will reexecute such collectives.

Summary of collectives

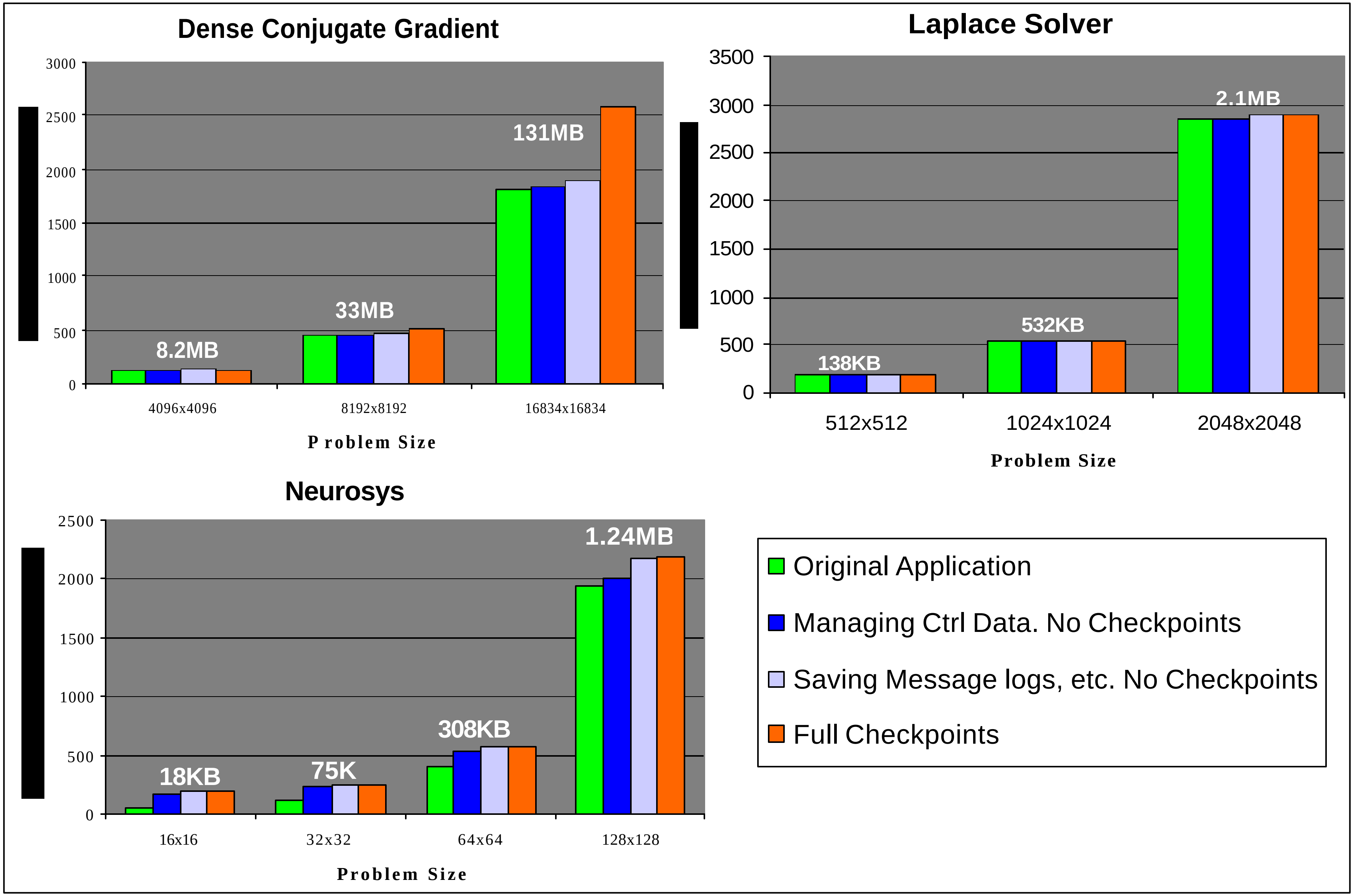
- Single-Receiver Collectives introduced.
- There are solutions for every type of collectives.
- Each solution works off of the same protocol platform but with different key choices.

- Result: a single protocol for all of MPI.

Implementation

- Implemented the protocol on the Velocity cluster in conjunction with a single-processor checkpointer.
- We executed 3 scientific codes with and without checkpointing.
  - Dense Conjugate Gradient
  - Laplace Solver
  - Neuron Simulator
- 16 processors on the CMI cluster
- Measured the overheads imposed by the different parts of our checkpointer.

Performance of Implementation



Contributions to Date

- Developed and implemented a novel protocol for distributed application-level checkpointing.
- Protocol can transparently handle all features of MPI.
  - Non-FIFO, non-blocking, collective, communicators, etc.
- Can be used as sand-box for distributed application-level checkpointing research.

Future Work

- Extension of application-level checkpointing to Shared Memory

- Compiler-enabled runtime optimization of checkpoint placement

  (Extending the work of CATCH)

- Byzantine Fault Tolerance

Shared Memory

- Symmetric Multiprocessors – nodes of several (2-64) processors connected by a fast network.
- Different nodes are connected by a slower network.
- Typical communication style:
  - Hardware shared memory inside the node
  - MPI-type message passing between nodes

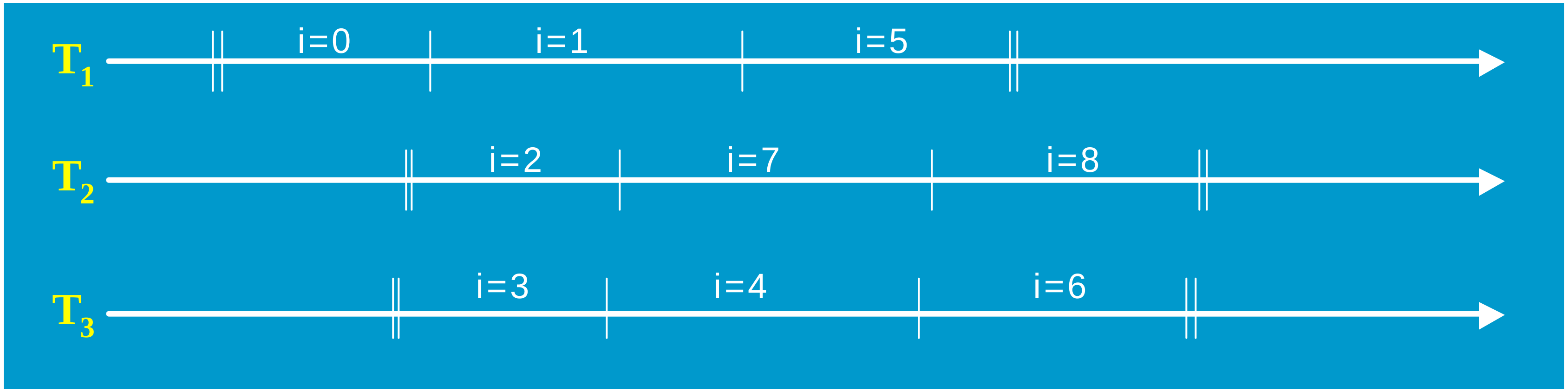
OpenMP

- An industry standard shared memory API.
- Goal: create a thin layer on top of OpenMP to do distributed checkpointing.
- Must work with any OpenMP implementation.

Issues with checkpointing OpenMP

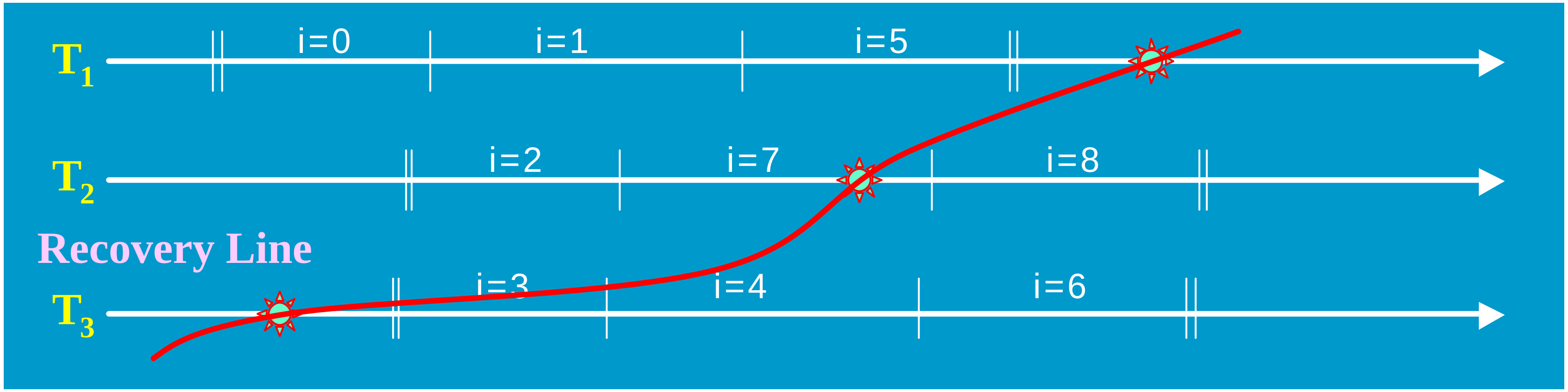
- Parallel for
  - different threads execute different iterations in parallel
  - iteration assignment is non-deterministic
- Flush
  - shared data that has been locally updated by different threads is redistributed globally
- Locks
  - carry only synchronization, no data

OpenMP – parallel for



- Different OpenMP threads execute different iterations in parallel.
- Iteration allocation is non-deterministic.

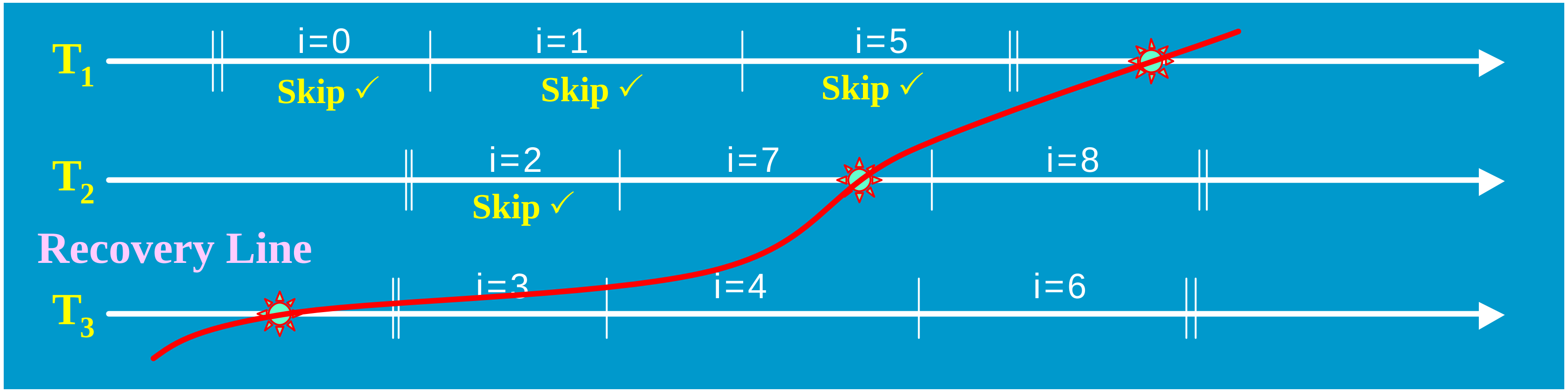
OpenMP – parallel for



- While executing a parallel for we keep track of which iterations we've completed.

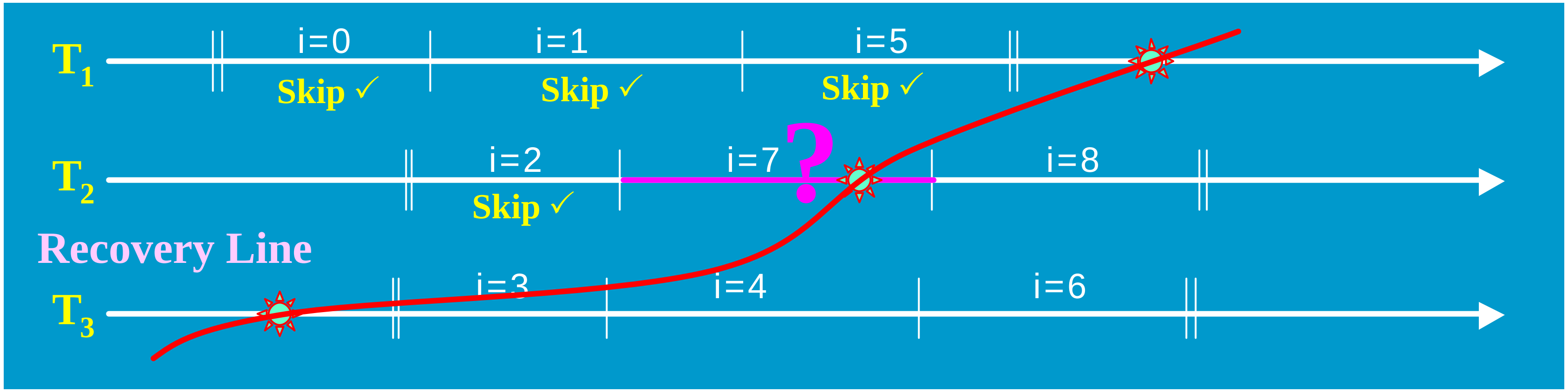
Above: [0,1,2,5] are completed
[7] is in progress

OpenMP – parallel for



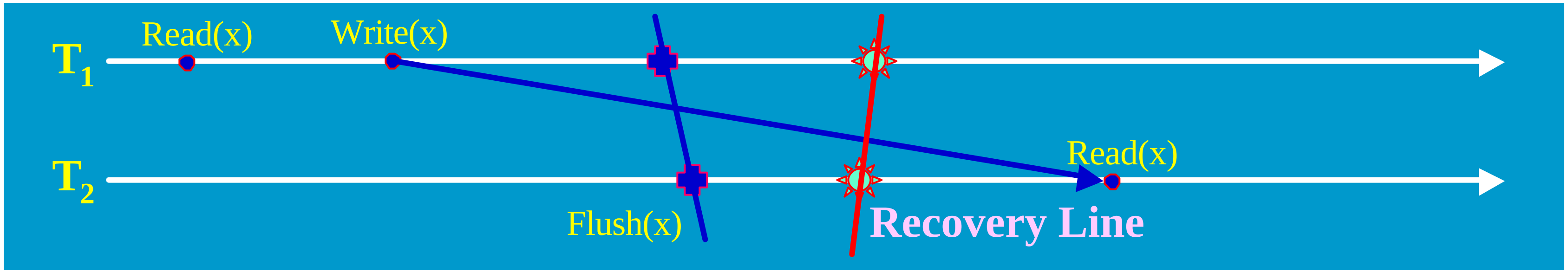
- If any thread in a recovery line checkpoints inside a parallel for, we must reexecute the parallel for.
- Iterations lying behind the recovery line are skipped by the threads that get them.

OpenMP – parallel for



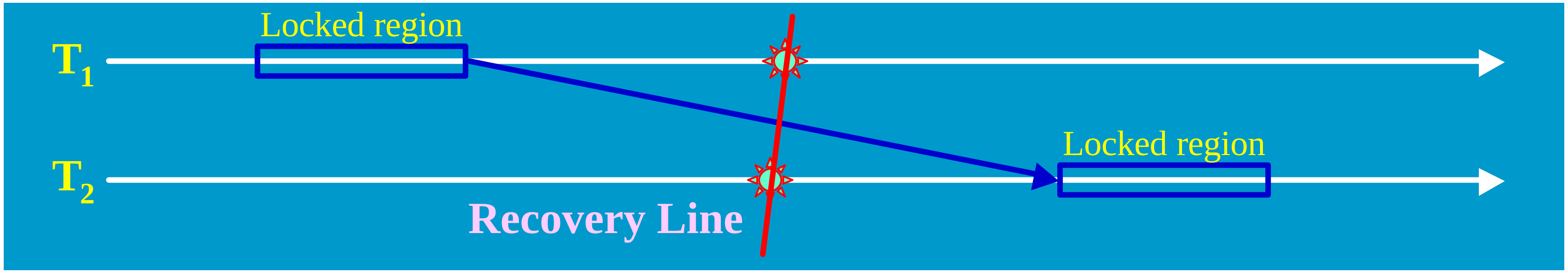
- Question: How we ensure that Thread 2 gets Iteration 7 on recovery?

OpenMP – Flush



- Flush(x) updates all threads to the current value of x. (last written by T_1)
- We can tread Flushes as data flows and use our MPI protocol.
- The above is a lot like a Late message.

OpenMP – Locks



- Locks are data flows that carry no data.
- This lock flow is trivial to enforce.
- Backwards lock flows are more complex.
- We cannot guarantee true synchronization wrt outside world.