Practical Scalable Consensus for Pseudo-Synchronous Distributed Systems

Thomas Herault[1], Aurélien Bouteiller[1], George Bosilca[1], Marc Gamell[2], Keita Teranishi[3], Manish Parashar[2], Jack Dongarra[1,4]

1 – University of Tennessee Knoxville
2 – Rutgers University
3 – Sandia National Laboratories
4 – Oak Ridge National Laboratories, Manchester University

Practical Scalable Consensus

Outline

Consensus

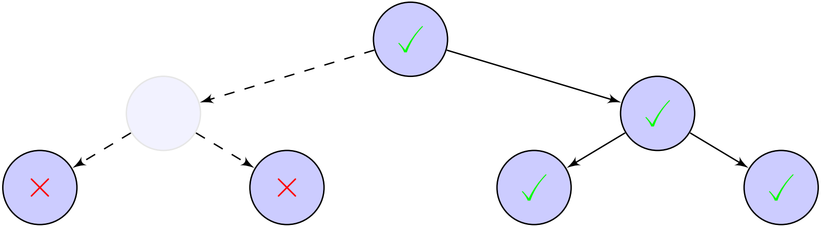
> *[consensus] is fundamental to distributed computing unreliable environments: it consists in* **agreeing** *on a piece of data upon which the computation depends*
>
> *M.Fischer, Brief Survey on Consensus*

D.Davies, J.F.Wakerly "*Synchronization and Matching in Redundant Systems*", IEEE Trans. on Comp., 1978. Context: Triple Modular Redundancy. Conclusion: Agreement through voting can tolerate only a minority of faulty processors.

Consensus is ubiquitous in distributed systems with high-availability (e.g. distributed database). It is a critical component in Fault-Tolerant HPC systems.

Consensus in the context of HPC

Consider the case of a broadcast implemented with a binary tree.



Failures, that happen during the execution, introduce inconsistencies: not all processes know that the broadcast operation failed.

Consensus (or agreement) allows to reconcile inconsistent / non-uniform states due to failures.

It must be reliable.

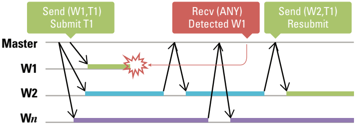
It must be efficient, especially in the failure-free case.

ULFM

ULFM provides targeted interfaces to empower recovery strategies with adequate options to restore communication capabilities and global consistency, at the necessary levels only.

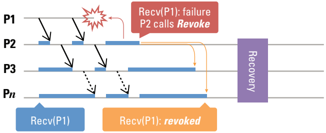
CONTINUE ACROSS ERRORS

In ULFM, failures do not alter the state of MPI communicators. Point-to-point operations can continue undisturbed between non-faulty processes. ULFM imposes no recovery cost on simple communication patterns that can proceed despite failures.



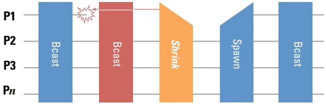
EXCEPTIONS IN CONTAINED DOMAINS

Consistent reporting of failures would add an unacceptable performance penalty. In ULFM, errors are raised only at ranks where an operation is disrupted; other ranks may still complete their operations. A process can use MPI_[Comm,Win,File]_revoke to propagate an error notification on the entire group, and could, for example, interrupt other ranks to join a coordinated recovery.



FULL-CAPABILITY RECOVERY

Allowing collective operations to operate on damaged MPI objects (Communicators, RMA windows or Files) would incur unacceptable overhead. The MPI_Comm_shrink routine builds a replacement communicator, excluding failed processes, which can be used to resume collective communications, spawn replacement processes, and rebuild RMA Windows and Files.



ULFM Agreement Specification

```
int MPIX_Comm_agree(MPI_Comm comm, int *flag);
MPIX_COMM_AGREE(COMM, FLAG, IERROR)
        INTEGER COMM, FLAG, IERROR
```

`comm` the communicator on which to apply the consensus

`flag` An in/out integer: in input, the process participation, in output, the result of the agreement on these ints (bitwise and)

`return value` An error code if new process failures were discovered during the agreement, or success

The operation implements an agreement on the couple (`flag`, `return code`): all surviving process, despite any failure have the same values in each (even if the return code is an error, flag is defined).

Specification

Correctness

Termination	Every living process eventually decides.
Integrity	Once a living process decides a value, it remains decided on that value.
Agreement	No two living processes decide differently.
Participation	When a process decides upon a value, it contributed to the decided value.

Traditional consensus relies on Validity
This is because one value is chosen.

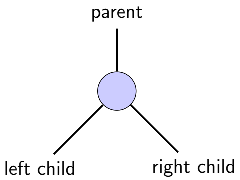
ULFM does not require the consensus to be uniform

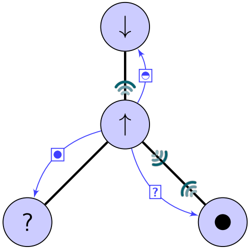
Assumptions

- Processes have totally ordered, unique identifiers
- Any process belonging to a group knows what processes belong to that group
- Any process may be subject to a permanent failure
- The network does not lose, modify, nor duplicate messages, but communication delays have *unknown* bounds
- The system provides a Perfect Failure Detector ($\mathcal{P}$):
  - All incorrect processes are eventually suspected by all correct processes
  - No correct process is ever suspected by any process
- The operation of the consensus is associative and commutative, and idempotent, with a *known neutral element*

Outline

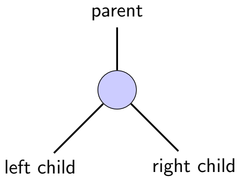
Principle and Notation

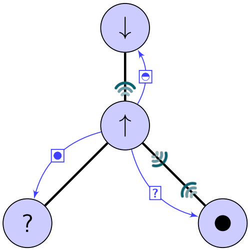




- Processes are arranged following a mendable tree topology: given a list of known dead processes, they communicate or monitor the liveliness of only their neighbors in that topology.
- The algorithm is a resilient version of Fan-in / Fan-out: all contributions (noted ◓) are reduced along the tree up to the root, that broadcasts it
- *Deciding* the result of the consensus for a given process consists in remembering the return value of the consensus, broadcasting it to the current children, and returning *as if* the consensus was completed.

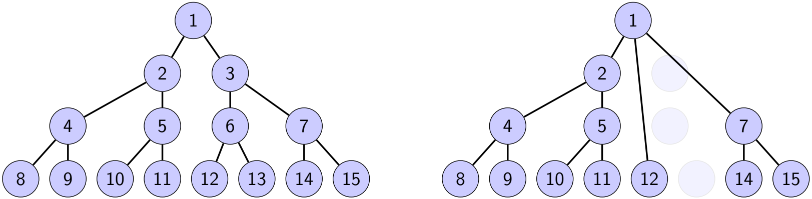
Principle and Notation





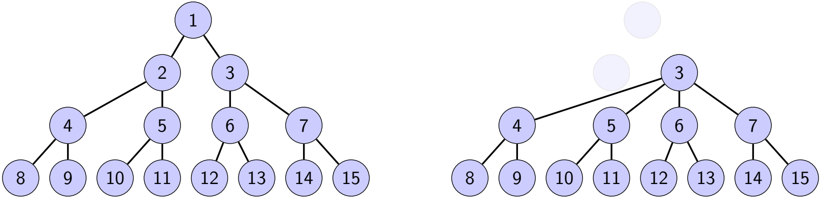
- Alive processes can be in 3 states:
  - ?, if they have not entered the consensus yet
  - ↑, if they are waiting from the contribution of their children
  - ↓, if they have sent their contribution to their parent and are waiting for the decision
  - ●, if they have received the decision
- There are 3 types of messages:
  - ◓, when a process sends its participation to a parent
  - ●, when a process broadcasts the decision to its children
  - ?, when a process enquired about a possible result of a completed consensus
- Processes can monitor ( ) other processes for failures

Mendable Tree for Consensus



The Fan-in Fan-out tree used during the consensus is mended, as failures are discovered during the execution.
The mending rule is simple: processes are arranged according to their (MPI) rank following a breath-first search of the tree, assuming no failure (left tree)

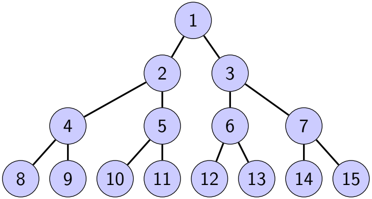
Mendable Tree for Consensus

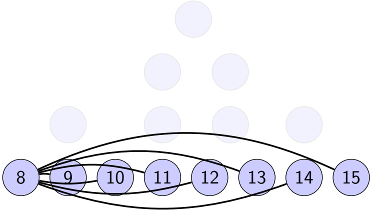


Nodes replace their parents by the highest-ranked alive ancester in the tree in case of failure.

Processes without an alive ancestor in the original tree connect to the lowest alive processor as their parent. *The lowest alive processor is always the root of the tree*

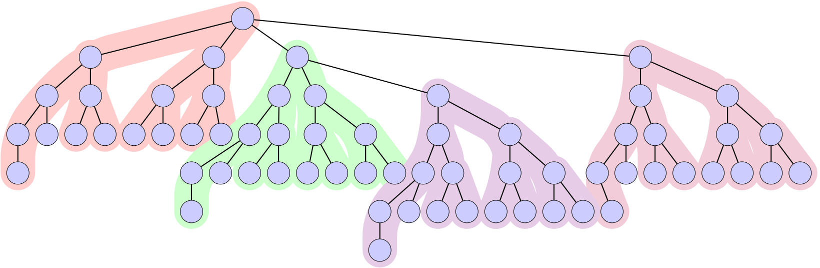
Mendable Tree for Consensus





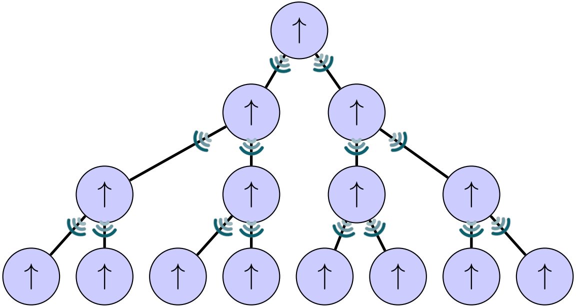
If half the processes die, the tree can, in the worst case, degenerate to a $np/2$-degree star

Architecture-Aware Tree



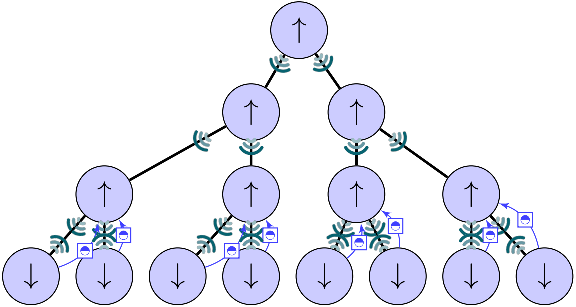
To map the hardware network hierarchy, two levels of trees are joined: In the example, *representative* processes of nodes (node0, node1, node2, node3) are interconnected following a *binary* tree, and processes belonging to the same node (16 process / node in this case) are also connected following independent *binary* trees.

No Failure



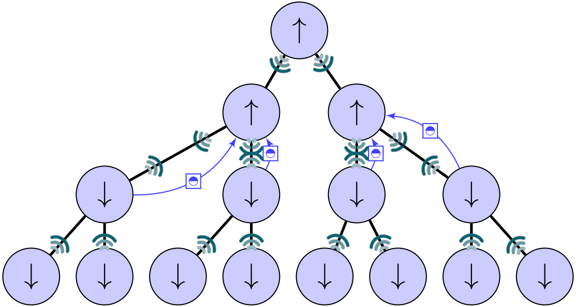
Initially, all processes are in the state ↑ to provide their participation, and the participation of their descendents to their ascendent. Each process monitors its descendents for possible failures (🔊) until they have participated.

No Failure



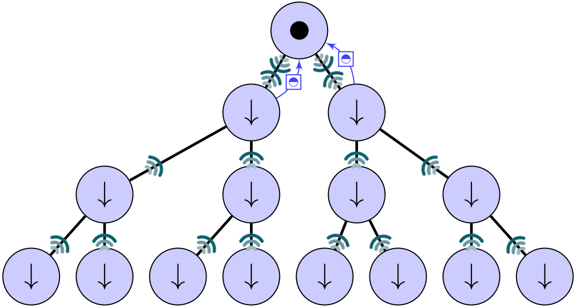
Leaves can send their participation (◓) to their parent, and enter the broadcasting state ↓. They start monitoring their parent for possible failures (🔊)

No Failure



Once a process has aggregated the participation of all its descendents, it can forward the information upward and do the same

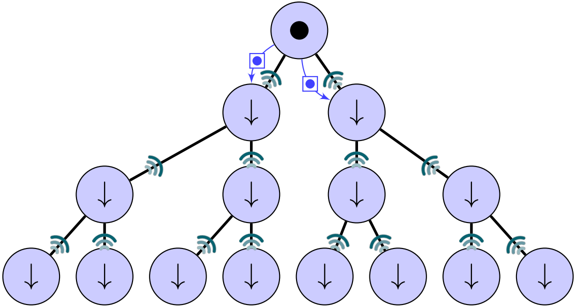
No Failure



Once a process has aggregated the participation of all its descendents, it can forward the information upward and do the same

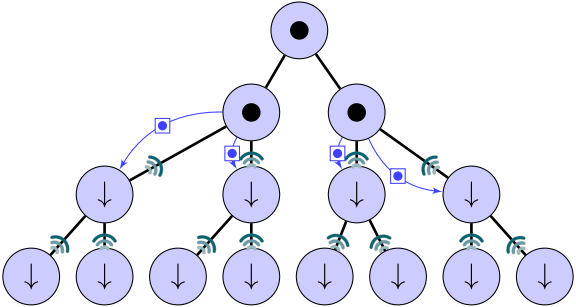
The root process can *decide* as soon as all descendents have contributed, it enters the decided state ●, starts broadcasting the decided message (●) to its descendents, and stops monitoring processes for failures

No Failure



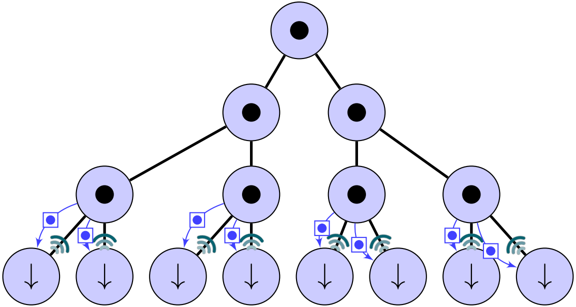
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

No Failure



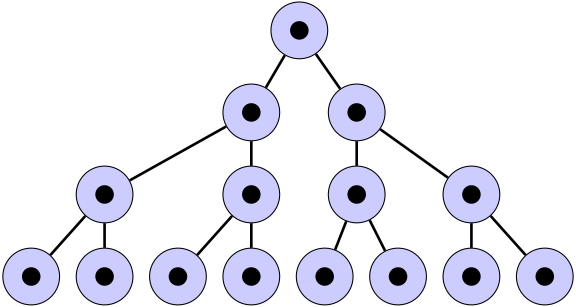
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

No Failure



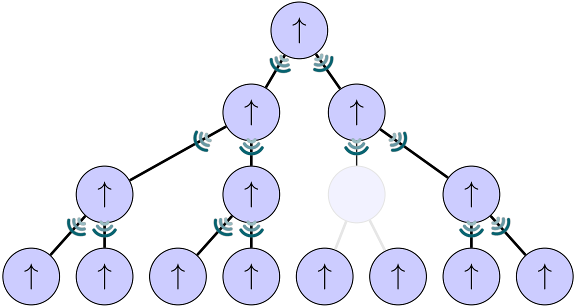
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

No Failure



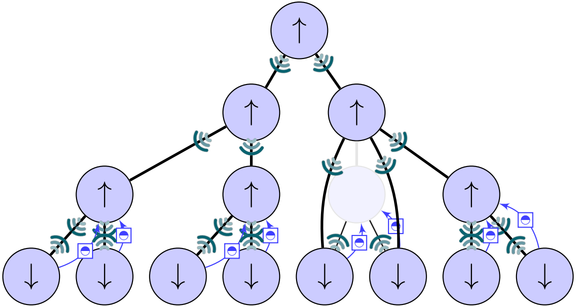
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

Failure before participating



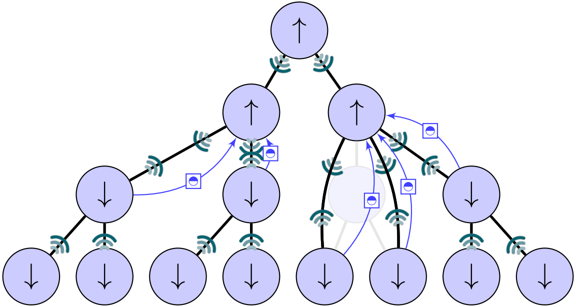
Process P_6 died before participating. P_3, its parent, starts monitoring it ( ) when it enters the consensus (state $\uparrow$).

Failure before participating



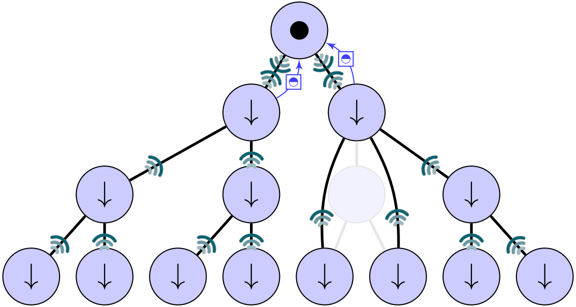
Processes P_{12} and P_{13} will send their participation (◓) to P_6, these messages are lost, and they start monitoring (🔊) P_6. P_3 eventually discovers the death of P_6, and starts monitoring (🔊) its new descendents P_{12} and P_{13}.

Failure before participating



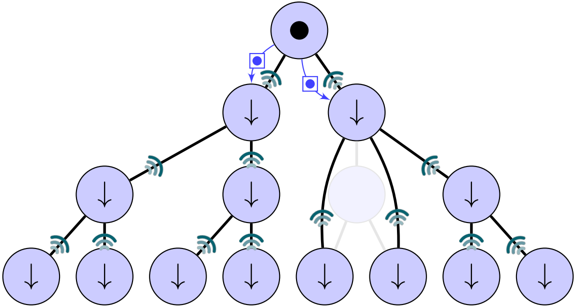
Processes P_{12} and P_{13} eventually discover the death of P_6, and take P_3 as their parent, sending it their participation (◓). They also start monitoring (🔊) their new parent, P_3.

Failure before participating



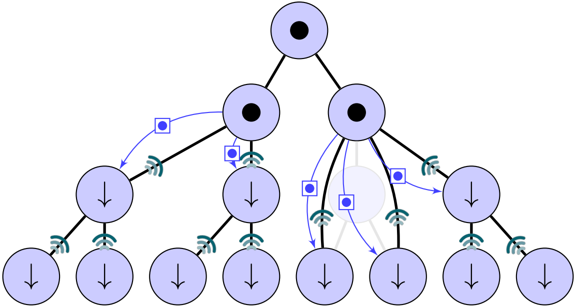
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



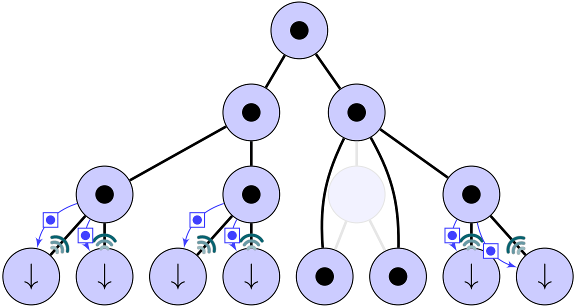
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



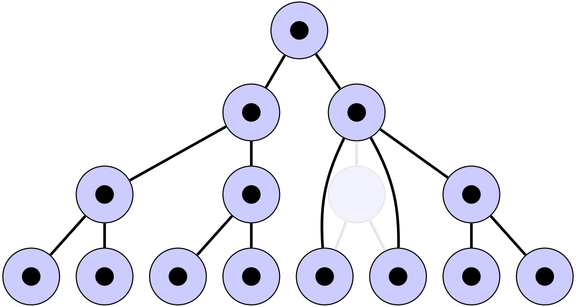
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



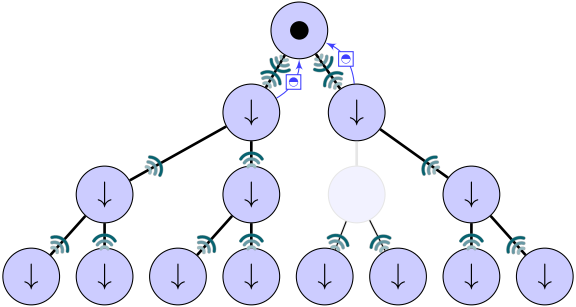
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



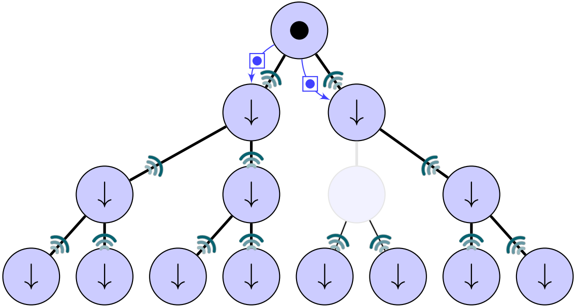
The tree being fixed, the information simply flows along the mended tree as initially.

Failure After Participating



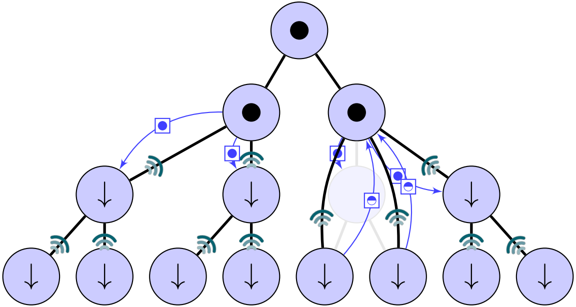
Process P_6 fails, but after participating to the current consensus.

Failure After Participating



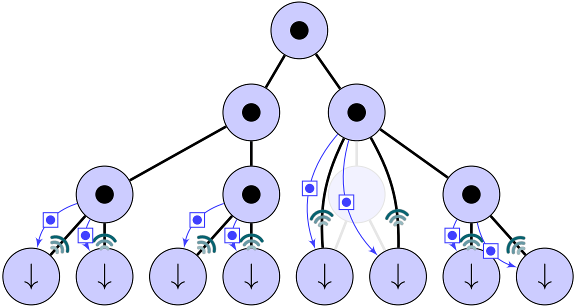
If it was a leaf, that would not prevent the consensus to complete. Since it has children, and they have not received the decision (●) yet, they are monitoring ( ) it, and eventually discover the death

Failure After Participating



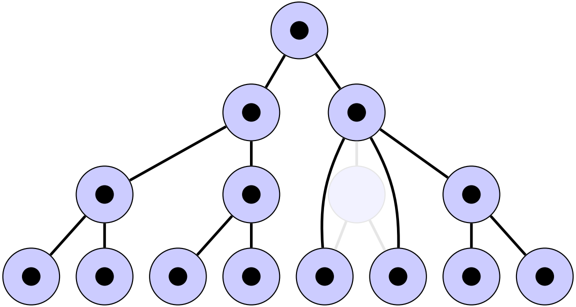
They send their participation (◓) back to their grand-parent, P_3, starting to monitor it (🔊). This ensure that if P_6 died before forwarding it upward, their participartion (◓) is not lost. This also reconnects the tree.

Failure After Participating



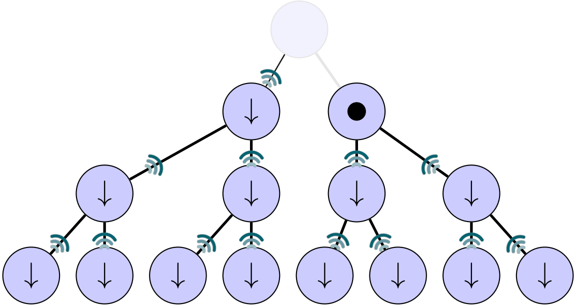
Even if P_3 is already done with the current consensus, it remembers the result (ERA property), and provides the result (●) again, allowing the information to continue flowing down the tree.

Failure After Participating



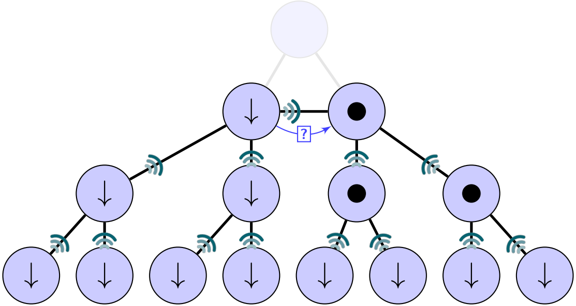
Even if P_3 is already done with the current consensus, it remembers the result (ERA property), and provides the result (●) again, allowing the information to continue flowing down the tree.

Failure of Root



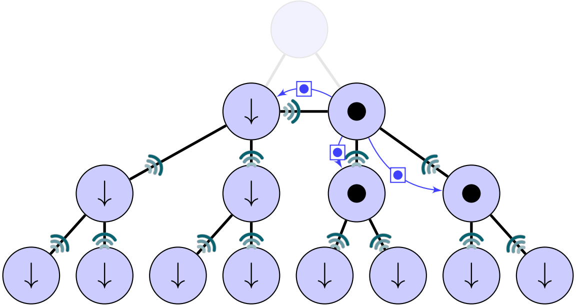
If the root of the tree dies after it started broadcasting the decision, but before it could reach all its children, the ones that did not receive the decision (●) are still monitoring that dead root ( ).

Failure of Root



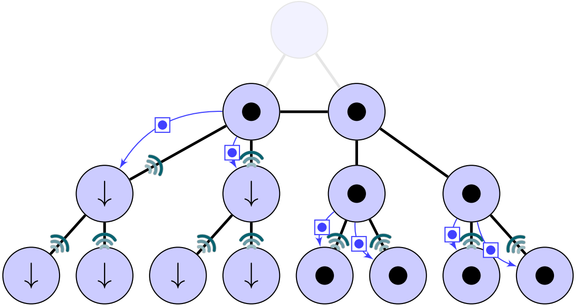
If a process becomes the root (lowest identifier), but was waiting for a decision, it asks all its new children if they received a decision before, by sending the message (?), and monitoring them ().

Failure of Root



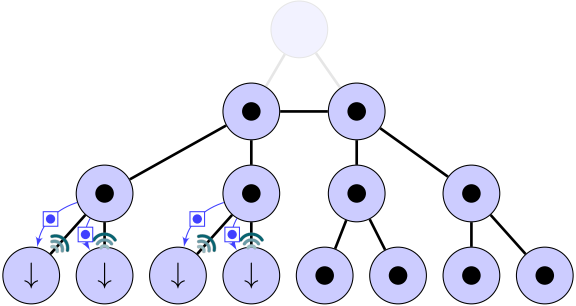
If one of them has the decision, it answers with it and the root can decide and broadcast (●). If none has it, they provide their participation (◓), if they reached that step, and wait for the decision of the new root.

Failure of Root



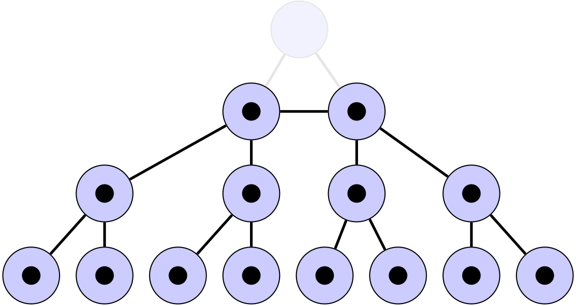
The broadcast of the decision (●) then continues along the tree

Failure of Root



The broadcast of the decision (●) then continues along the tree

Failure of Root



The broadcast of the decision (●) then continues along the tree

Implementation

Agreements are identified by a tuple (*CID*, *CEPOCH*, *ANUMBER*):

CID is the communicator Identifier

CEPOCH Epoch of the communicator – Epochs are changed every time a new communicator is created, and reflect how many failures were known at the time of creation

ANUMBER is the sequence number of the current agreement.

Current values of the agreements, progress status, and past values of past agreements are stored in hash tables.

The ERA is implemented at the *BTL level*, below the matching and message layer mechanisms.

Garbage Collection

When multiple consensus are executed on the same group of processes, processes executing ERA need to remember each consensus result. This can lead to memory exhaustion.

ERA implements a Garbage Collection mechanism to forget past consensus that *will not* be requested in the future.

That mechanism is implemented using the consensus operation itself: in addition to the consensus value, processes agree in the ● message on past consensus that can be collected.

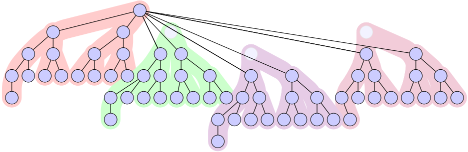
How to cleanup?

The last consensus is cleaned up by introducing an asynchronous ERA in the destructor of the communicator.

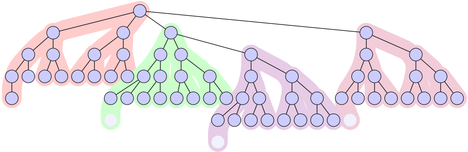
The result of this last ERA does not need to be remembered: if the communicator has been released, then all processes participated, and the return value is ignored.

Tree-Rebalancing

As processes crash, the Fan-in / Fan-out tree used to implement the two phases of the consensus can become unbalanced.



To implement the ULFM specification, all processes must agree on a list of failed nodes. Trees can be re-balanced when starting a new agreement based on that information.

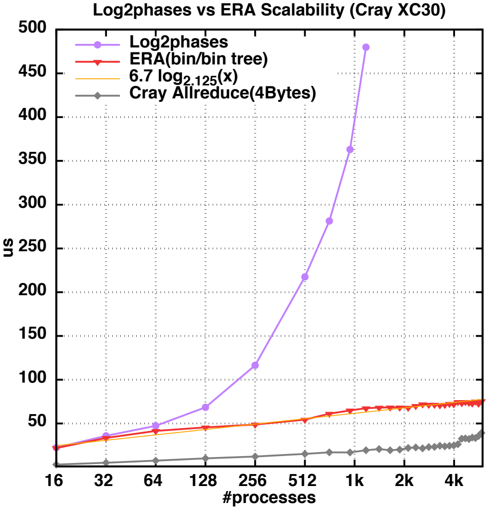


Outline

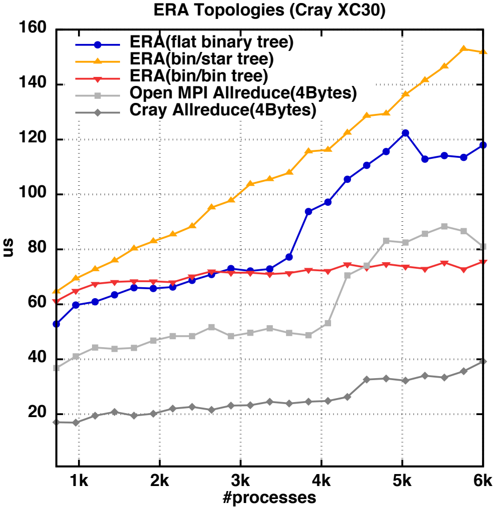
Environment

- NICS Darter: Cray XC30 (cascade)
  - `ugni` transport layer, with Aries interconnect
  - `sm` transport layer for shared memory
  - Scalability runs: 16 - 6,500 processes
- Benchmark:
  - `MPIX_COMM_AGREE` in loop
  - Measure duration:
    - before failure
    - during failure
    - stabilizing after failure
    - after stabilization

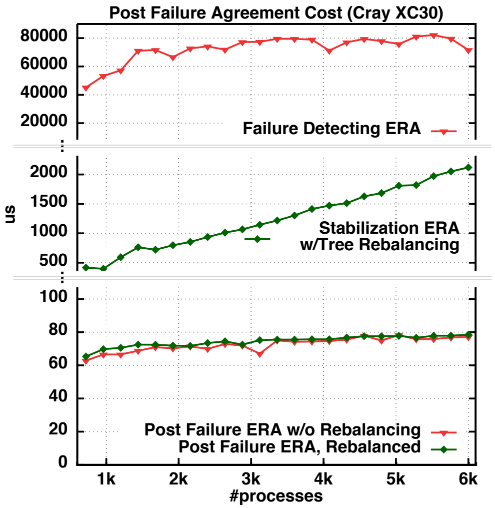
Agreement scalability in the failure-free case



ERA performance depending on the tree topology



Post Failure Agreement Cost



S3D and FENIX

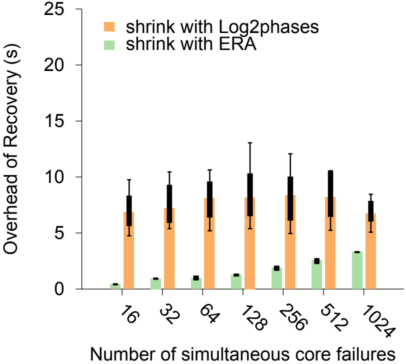
S3D

- Highly parallel method-of-lines solver for partial differential equations
- first-principles-based direct numerical simulations of turbulent combustion
- ported to all major platforms, demonstrates good scalability up to nearly 200K cores,

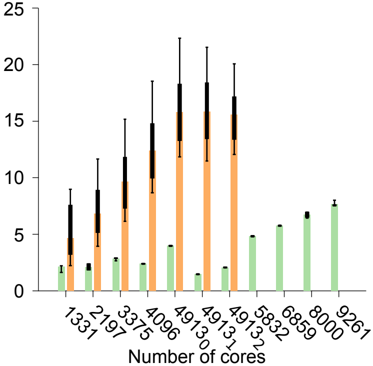
FENIX

- Online, Transparent recovery framework
- Encapsulates mechanisms to transparently
  - capture failures through ULFM return codes,
  - re-spawn new processes on spare nodes when possible,
  - fix failed communicators using ULFM capabilities,
  - restore application state, and return the execution control back to the application

FENIX & S3D Performance



Simultaneous failures on an increasing number of cores, over 2197 total cores



256-cores failure (*i.e.*, 16 nodes) on an increasing number of total cores

MiniFE and LFLR Framework

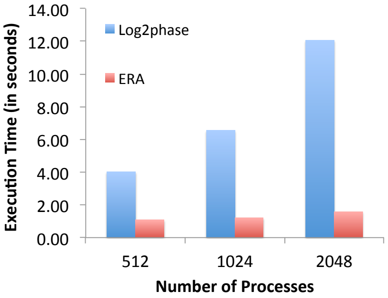
MiniFE

- Part of Mantevo mini-applications suite
- MiniFE performs a linear system solution with relatively quick mesh generation and matrix assembly steps.
- Modified version: performs a time-dependent PDE solution, where each time step involves a solution of a sparse linear system with the Conjugate Gradient (CG) method

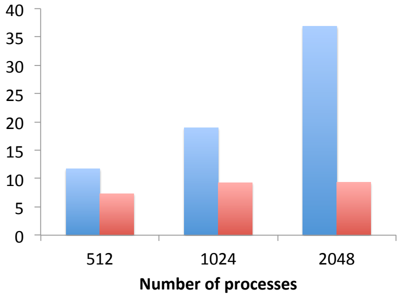
LFLR Framework

- Local Failure Local Recovery is a resilient application framework
- leverages ULFM to allow on-line application recovery from process loss without the traditional checkpoint/restart
- layer of abstraction classes to support `commit` and `restore` methods
- Works with active spare processes pool

MiniFE and LFLR Performance



Process and communicator recovery



Global agreement during 20 time steps.

Outline

Conclusion

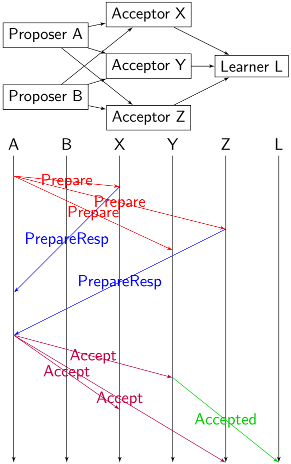
Summary

- ERA is a Logarithmic Agreement, in number of messages and in computation
- ERA allows processes to return early from the routine itself, serving potential late requests in the background
- Its implementation in ULFM / Open MPI shows performance comparable to an optimized non-fault-tolerant AllReduce
- Improvement of agreement translates into improvement of other routines (`shrink`).

Future Work

- Failure Detection is the next performance bottleneck
- ERA relies on perfect failure detection ($\mathcal{P}$)
- Implementing a low-latency / low-probability of false positive failure detector is a challenge

Why not use Paxos?



- PAXOS provides reliability in persistant environments (intermittent failures and persistent storage space; message loss and dupplication)
- It relies on replication of information: requests are sent to multiple processes, and a majority must acknowledge
- Given our different requirements, we can achieve lower latencies in the failure-free case,
- Decision in PAXOS is upon one proposed value, while we need a combination of proposed values

Multiple Phase Commit Agreements

"Scalable distributed consensus to support MPI fault tolerance":

- Three Phase Commit:
  - Ballot number is chosen
  - Value is proposed
  - Value is committed
- Reliable P.I.F. ($O(log_2(n))$ comm., $O(1)$ comp.)

"A Log-scaling Fault Tolerant Agreement Algorithm for a Fault Tolerant MPI":

- Two Phase Commit
  - Fan-in / Fan-out approach
- Fatal errors when the root dies during the agreement
- $O(log_2(n))$ comm., but $O(n)$ comp.