

Fault-Tolerance? No Problemo.

Jeff Napper,
Lorenzo Alvisi, Calvin Lin, Alison Smith
The University of Texas at Austin

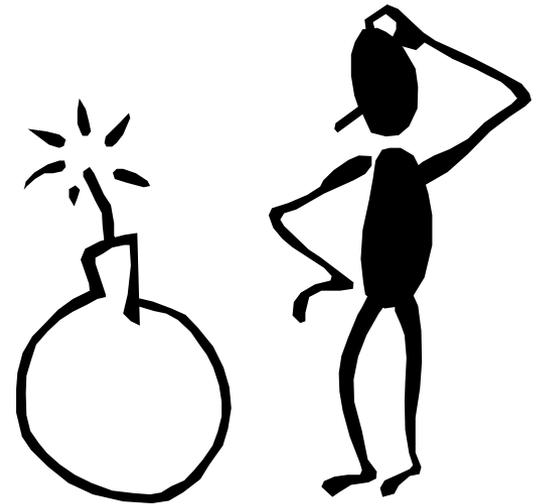
Fault-Tolerance: The Past...

- Life-critical applications
 - Tolerate arbitrary failures
 - Small systems (10^1)
 - Goal: Continuous availability
- Cost is secondary concern
 - Expensive resources
 - High overhead



The Present...

- Mission-critical applications
 - Tolerate crash failures
 - Medium systems (10^2)
 - Goal: Fast recovery
- Cost is important factor
 - Lower overhead
 - Passive replication



The Future

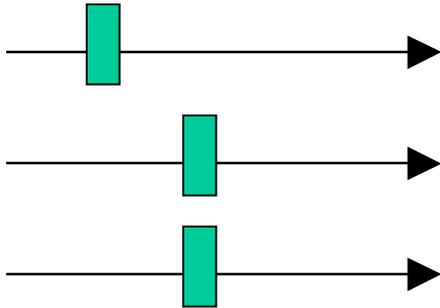
- Large applications
 - Tolerate crash failures
 - Large-scale systems (10^4)
 - Goal: Shorter time to completion
- Efficiency is important factor
 - Lower costs
 - Increase scalability



Outline

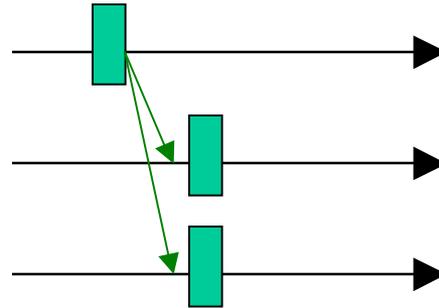
- Fault-tolerance at a glance
- Current work — Egida
- Future Work
 - Implicit Coordinated Checkpointing
 - Scalable Message Logging
- Conclusions

Flavors of Distributed Checkpointing



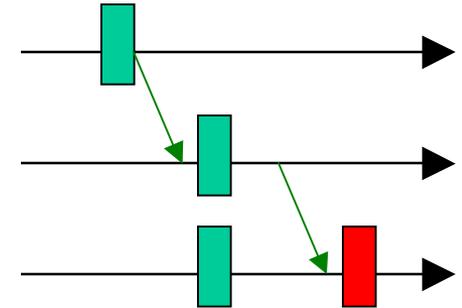
Independent

- ✓ Simplicity
- ✓ Autonomy
- ✓ Scalability
- ✗ Domino Effect



Coordinated

- ✓ Consistent States
- ✓ Good Performance
- ✓ Garbage Collection
- ✗ Scalability



Communication-induced

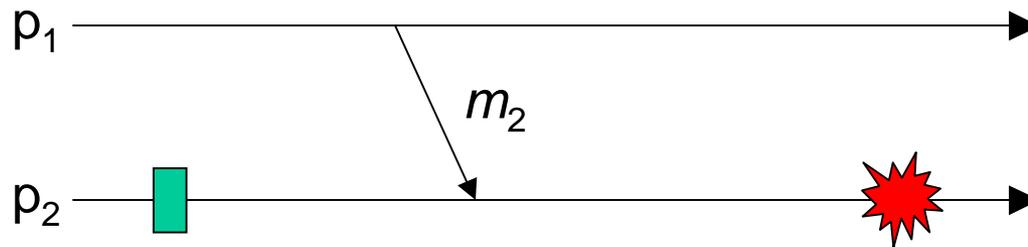
- ✓ Consistent States
- ✓ Autonomy
- ✓ Scalability
- ✗ Performance

Unifying Theme

- All checkpointing protocols enforce global consistency
- The challenge is to ensure progress of the consistent cut
 - Coordination reduces the number of useless checkpoints

Message Logging at a Glance

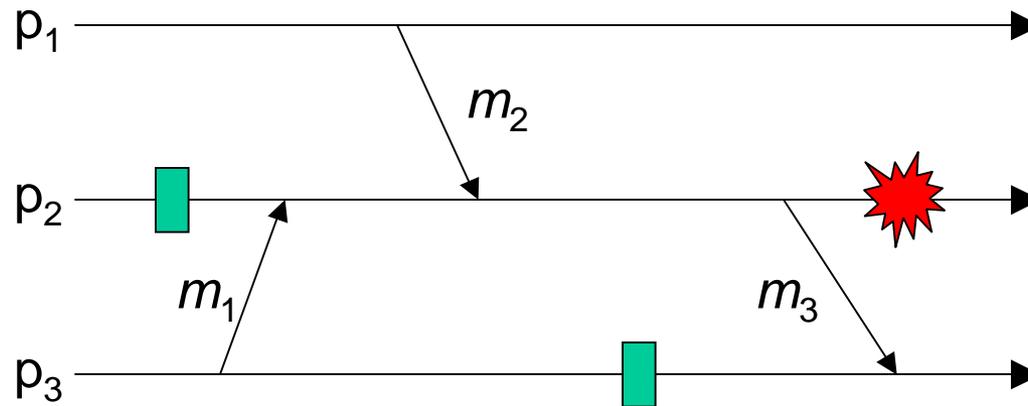
- Log information on stable storage during failure-free execution
- Use that information to recover from failure



Orphan: process that depends on an unrecoverable state of a failed process

Piecewise determinism: all non-deterministic events can be identified and logged in the event's *determinant*

Flavors of Message Logging



Pessimistic

- ✓ No orphans
- ✓ Easy recovery
- ✗ Blocks

Optimistic

- ✓ Non-blocking
- ✗ Orphans
- ✗ Complex recovery

Causal

- ✓ Non-blocking
- ✓ No orphans
- ✗ Complex recovery

Unifying Theme

- All message logging protocols enforce the no-orphans consistency condition
- The challenge is managing non-determinism
 - A process may interact with other processes or with the environment, generating dependencies on these events
- Characterize a protocol according to how it handles non-determinism
 - Identify relevant events
 - Specify which actions to take when an event occurs

Egida

- **Transparent**
 - Provides seamless integration with applications
- **Extensible**
 - Easily handles new sources of non-determinism
 - Easily includes new protocols
- **Flexible**
 - Allows developer to select best protocol
- **Powerful**
 - Allows experiments with different protocols to assess costs

Events in Egida

- Non-deterministic events
 - message delivery, file read, lock acquire
- Failure-detection events
 - timeout, message delivery
- Internal dependency-generating events
 - message send, file write, lock release
- External dependency-generating events
 - output to printer, screen, or file
- Checkpointing events
 - timeout, explicit invocation, message delivery

The Architecture

- Event handlers invoked on relevant events
- Library of modules
 - Implement core functionalities:
Checkpointing, managing determinants, logging, piggybacking, detecting orphans, restarting process
 - Provide basic services
Stable storage, failure detection, etc.
 - Single interface for multiple protocols

Protocol Specification

- Use a specification language to select desired modules (at compile-time) corresponding to implementations

/ non-deterministic events statement */*

receive:

determinant: {source, ssn, dest, desn}

Log: determinant on volatile memory of processes

/ checkpoint statement */*

*Checkpoint: independent,
asynchronous on NFS disk*

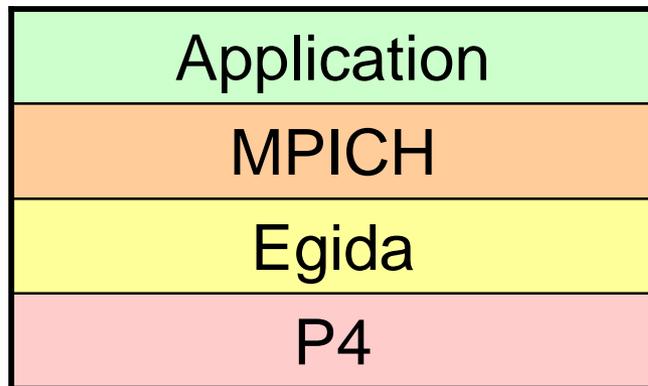
Implementation: incremental

Scheduling policy: periodic

- Synthesize protocol automatically from specification

Integration with MPI

- MPICH:
 - 2-layered architecture
 - Upper layer exports MPI functions to application
 - Lower layer performs data transfer using platform specific libraries



- Modifications to MPICH:
 - In upper layer, replace calls to P4 (send/recv) with corresponding calls to Egida API
- Modification to P4:
 - Handle socket-level errors
 - Allow recovering process to set up connections with correct processes
- Modifications to Applications:

NONE

Current Status

- Porting Egida from Solaris to Linux
- Specializing Egida for MPICH
 - Use tag information to determine communication structure

Scalable Fault-Tolerance?

- Current approaches do not scale
 - Coordinated checkpointing requires all processes to write to stable storage nearly simultaneously
 - Message logging requires logging all determinants and performing broadcasts during recovery
- **Assumption: We cannot use master/slave model for all computations**

Rules of Thumb

- Minimize use of stable storage
 - How much data is written? How often?
 - How to store process updates cheaply?
- Compute cycles are cheap
 - Trade computation for either communication or stable storage?
- Communication is often structured
 - How to exploit structure?

Implicit Coordinated Checkpointing

- Efficient checkpointing is required for *any* passive replication solution
- **Assumption:** Typical coordinated checkpoints (Chandy-Lamport) are very expensive
 - ✗ Requires nearly simultaneous ckpts
 - ✗ Limited by file system bandwidth
- **Solution:** Schedule checkpoints to use available bandwidth
 - ✓ Simple communication structure allows compiler to build message dependency graph
 - ✓ Compiler statically assigns time to checkpoint to build consistent cut

Scalable Message Logging

- Adapt protocols to structured communication
- Message logging may be cheaper than high-frequency checkpointing
- **Assumption:** The aggregate size of messages sent is smaller than ckpts
 - ✗ Checkpoints include all modified state
 - ✗ Overhead incurred per message
- **Solution:** Scale message logging using hierarchical structures
 - ✓ Allows smaller recovery groups to eliminate broadcasts
 - ✓ Use cheap compute cycles to recreate state from messages

Conclusions

- Try simplest approach first!
- Coordinated checkpointing is the simplest form of fault-tolerance
 - Can we make it scalable?
 - Is the overhead low enough?
- Egida allows us to explore rollback recovery protocols
 - Is the overhead low enough?