Resource Management (3)

Grid Computing (M)

Lecture 13

UNIVERSITY
of
GLASGOW

Lecture Outline

- Reliability of a computational grid
- Failure modes
- Fault tolerance
  - Checkpoint and retry
  - Avoiding systematic software failures
    - Error detection
    - Damage confinement
    - Techniques for error recovery
  - Implications of scale and autonomy
  - Economics

Reliability of a Computational Grid

- Grid computing ⇒ Large, multi-organisational, heterogeneous distributed systems

- p(failure) ∝ size of system
- Failure is inevitable due to environmental issues
  - Even if the software implementation is perfect; hardware is not
  - Any sufficiently large system will exhibit partial failures

- Need to ensure consistent results in presence of failures
- Critical to understand failure and recovery modes

Failure Modes

- What can fail?
  - Processes
  - Communication channels
  - Storage devices
  - ...
- How can it fail?
  - Omission failure
    - Fail-stop; clean halting failure, stays failed; detectable (e.g. closes connections)
    - Crash; unclean halting failure, stays failed; not necessarily detectable
    - Omission; message sent but vanishes
  - Arbitrary failures
    - Incorrect results, message corruption, intermittent lack of response
  - Timing failure
    - Results are late, fail to meet timing deadline

Failure Modes

- All practical systems exhibit arbitrary failure modes
  - Network may corrupt, delay, reorder, discard or duplicate packets
  - Processes can generate incorrect or arbitrarily late answers
    - Buffer, array bounds, or stack overflows
    - Lack of synchronisation in multithreaded systems
    - Incorrect algorithms
    - Arithmetic overflows
    - Infinite loops
  - Storage may fail
    - Disks may corrupt blocks
    - File system may corrupt or lose data
  - Processors or memory can fail
    - Bugs in processor design/implementation
    - Radiation (α-particle) damage to memory
    - Electrical noise

Fault Tolerance

- To build a robust computational grid, need to tolerate failures

- Several levels to this:
  - How to tolerate partial failure within a node?
    - Checkpoint and retry
    - Atomic transactions and recovery blocks
    - *N*-version programming
    - Retransmission/repair of lost data
  - How to tolerate partial failure of a distributed system?
    - Checkpoint and retry
    - Distributed transactions
    - Two-phase commit
  - Implications of autonomy

Checkpoint and Retry

- Simplest approach: periodically checkpoint process state, retry execution on failure
  - Widely used in grids and cluster computing (e.g. condor)

    [Details in last lecture]
- What does this protect against?
  - Hardware failures? Yes
  - Software failures? Maybe
    - Does *not* protect against systematic failures in the process
      - An incorrect algorithm will give same answer on retry
      - A buffer will always overflow given the same input data
    - Might protect against transient failures
      - Synchronisation problems and race conditions, since timing will differ on retry
      - Failures due to external component, when external component has recovered
- Need to avoid infinite retry loops
  - Exponential back off with eventual timeout?

Avoiding Systematic Failure

- How to protect against systematic failures?
- Run-time analysis of system correctness
  - Error detection
  - Damage confinement
  - Error recovery
  - Fault tolerance and continued service

- Must program defensively
  - Use techniques from safety critical systems world to improve reliability, availability and to validate correctness
  - Currently unusual in grid computing
    - More typical just to terminate faulty jobs
    - But might be *required* in certain fields
      - e.g. computational medical research, simulations for aircraft design

Error Detection

- Environmental detection
  - Illegal instruction, segmentation violation, floating point exception, array bounds exception
  - Trivial to detect with modern languages and operating systems
- Application detection – acceptance test for results
  - Replication checks
    - Different algorithms – should get the same answer
  - Reversal checks
    - If one-to-one relationship between input and output, reverse calculation and see if you get the correct input based on the output
  - Coding checks
    - Error correction codes (e.g. parity, Reed-Solomon) can detect corrupt data
  - Reasonableness checks
    - Is the result within sensible bounds?
    - `assert()` in C/Java; invariants, pre- and post-conditions in Eiffel
    - E.g. a weather forecasting system predicting 900 m.p.h. winds is likely faulty

Damage Confinement

- Once error detected, want to limit propagation
- Decompose system into atomic actions
  - Succeed or fail; don't expose partial results
  - Needs a well-defined boundary of the action
    - Lock any needed resources
    - Perform calculation using only locked resources, local variables
      - Easy to leak information, since no language support
      - Difficult to ensure clean rollback of state on failure
    - Unlock resources
  - Widespread support in databases; little in general purpose systems
    - Wide interest in *Software Transactional Memory* in Haskell community
  - Same approach works across multiple hosts – a distributed transaction – if there is distributed scheduler support to coordinate locking and/or rollback between hosts

Atomic
Consistent
Isolated
Durable

Two-phase Commit Protocols

- How to agree results of a distributed transaction?
  - Nodes may fail at various points; messages may be lost

- Two-phase commit protocol:
  1. The commit manager assembles solicits votes on result
  2. Hosts reply with a local *commit* or *abort* message and wait
     - Hosts that vote *commit* guarantee they can complete action at later date
  3. The commit manager collects replies, decides on basis of vote
  4. The commit manager propagates a global *commit* or *abort* to all nodes
     - On receipt of message, nodes finalise state: commit or rollback and abort

- Eventually robust, provided all nodes use atomic actions, have appropriate timeouts for recovery
  - May need to retransmit messages on node failure/recovery

Error Recovery

- Several approaches to error recovery once damage confined
  - Erroneous computation:
    - Rollback
    - Recovery blocks
    - *N*-version programming
    - ...
  - Erroneous data transfer:
    - Forward error correction
    - Reliable multicast
    - ...

- Issues:
  - Local *vs*. distributed systems
  - Temporal *vs*. spatial recovery
  - Coordination of tasks and agreeing on results

Recovery Blocks

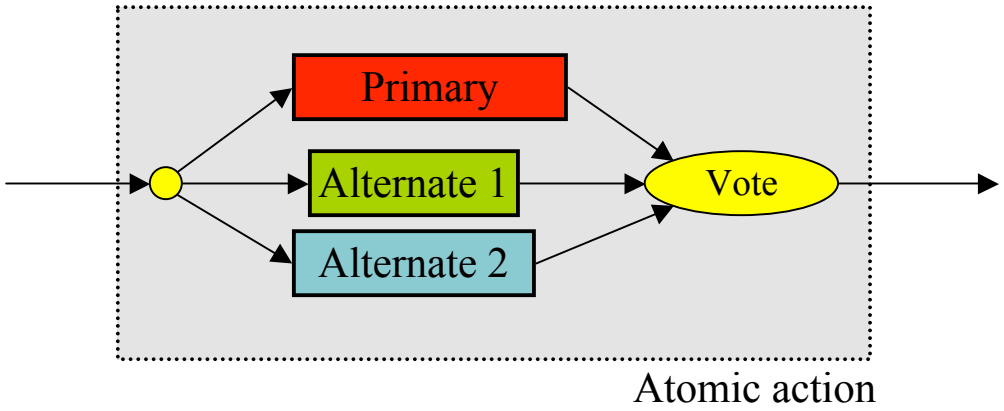
- Ensure correct computation by composing atomic actions into recovery blocks

```
ensure <acceptance test>
by
    <primary module>
else by
    <alternate module>
else by
    ...
else
    error
end
```

(can simulate the concept using exceptions)

- Relies on ability to implement multiple algorithms
  - Design and implementation diversity
  - What if there is only one way of solving the problem?

N-version Programming



- Alternative, but similar, approach: run the versions in parallel
  - The versions can be identical: protect from hardware failure

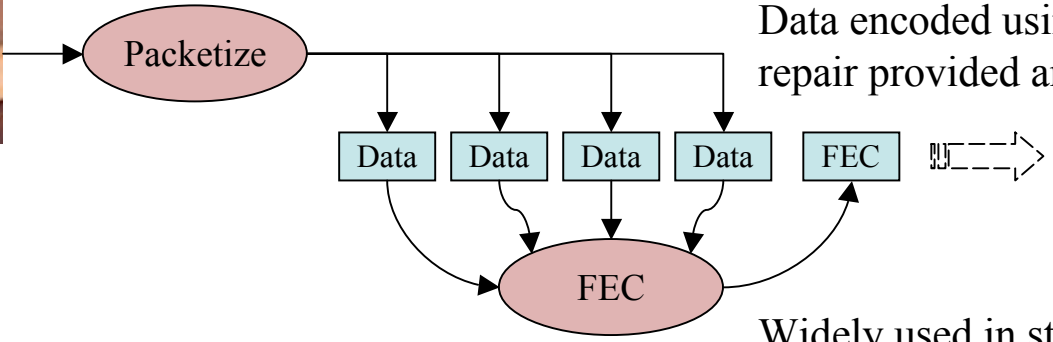
- Typically used for real-time control systems (e.g. Airbus)
  - Local coordination and voting

...but you have a massively parallel computational grid – use it!

Reliable Multicast Protocols

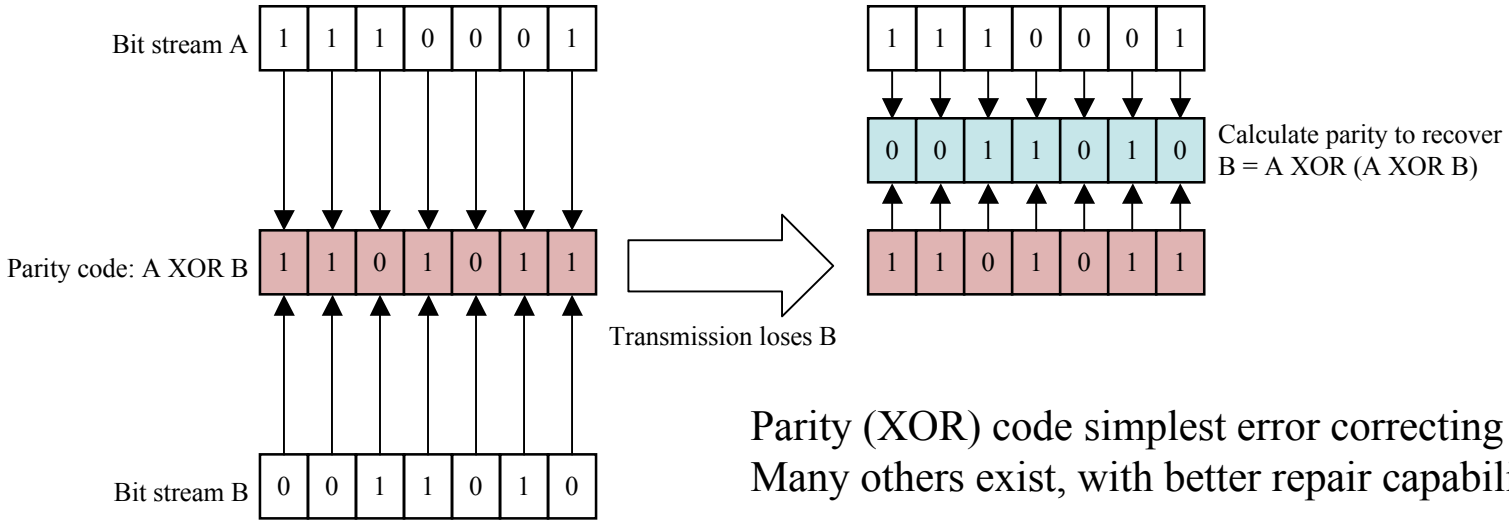
- How to recover erroneous/missing data?
  - Unicast retransmission using TCP well understood:
    - Issues with scaling to large bandwidth-delay product networks
      - [See "Scalability & Heterogeneity lectures" later in course]
    - Issues with reliability of checksums for large transfers
      - Between 1 packet in 1100 and 1 packet in 32000 fails TCP checksum due
      - Roughly 1 packet in 16 million has undetectable error ⇒ error every 16 gigabytes
      - [Stone and Partridge, "When the CRC and TCP Checksum Disagree", ACM SIGCOMM 2000]
  - Multicast repair more complex:
    - How to request repair?
      - How to avoid request storms?
      - As group size increase p(some receiver loses packet x) → 1.0
    - How to send retransmissions?
      - How to avoid implosion?
      - Error correcting codes useful; repair multiple errors with one packet
    - Solution outline: multicast everything, use scalable back-off triggered on group size/distance from requester

Error Correcting Codes



Data encoded using FEC: $n + k$ packets sent, repair provided any set of n packets received

Widely used in streaming media, in reliable multicast, and in peer-to-peer systems (e.g. used in OceanStore for robustness to loss of some copies of each block)



Parity (XOR) code simplest error correcting code. Many others exist, with better repair capabilities.

Fault Tolerance

- These techniques allow a large degree of fault tolerance
  - High cost: programmer time; execution time and data overheads
  - Cannot conceal all failures

- Can you continue service when part of the system fails and that failure can't be concealed?
  - At what point do you stop trying to recover and fail?
  - How much work can be salvaged when system fails?

Implications of Scale and Autonomy

- Implications of autonomy:
  - Service providers want to hide problems
  - Difficult to debug such distributed faults

- Traditional fault tolerance applied to grid computing systems:
  - Checkpoint and restart widely used
    - Relatively simple to implement
    - Rollback to beginning of job can often be done transparently to application
  - *N*-version programming relies on coordinated scheduling of multiple jobs, communication for voting; requires high programmer effort
    - More coordination than typically available in computational grids
  - Recovery blocks require high programmer effort; uncommon

  - Simple mechanisms widely used; more complex techniques available if needed in future

Economics of Fault Tolerance

- Is it worthwhile to implement fault tolerance?

- High programmer cost implementing recovery block or *N*-version programming
  - Require multiple algorithms and implementations
  - Can easily triple amount of design and coding work needed
  - Is that a good trade-off *vs*. debugging a single implementation?
  - What is the best way to prove correctness of implementation?
    - E.g. if grid computation is helping design a safety critical system, might want multiple algorithms and implementations, *in the way done for traditional real-time safety critical systems design*

- Checkpoint and restart cheap on programmer time
  - But may need to buy more grid resources, for restarted jobs
  - Although grid service provider might do this for free, to hide their failures!

Summary

- Reliability of a computational grid
- Failure modes
- Fault tolerance
  - Checkpoint and retry
  - Avoiding systematic software failures
    - Error detection
    - Damage confinement
    - Techniques for error recovery
  - Implications of scale and autonomy
  - Economics
- Current grids make widespread use of checkpoint and restart of long-running computational jobs
  - Protection from hardware failures
  - Minimal programmer effort
- More advanced techniques currently not widely used
  - Design diversity and security issues will become important to reliability as grids used for more critical tasks
  - Reliable multicast protocols will be important as grid computing adopts peer-to-peer protocols, multicast

Reminder: lecture tomorrow in Maths 325; Friday and next week tutorials in Kelvin Building, back in F121 on 27th February