Distributed OS

Hermann Härtig

Parallel Systems Software, short overview → MosiX

TECHNISCHE UNIVERSITÄT DRESDEN

17/05/11

Linux, Small kernels, and Linux

SMP (Linux, K42, …)

- Shared Memory SMP
- Linux syscall interface
- Balance Load, Optimise locality and concurrency

MPP (Tianhe-1A, Jaguar, Blue Gene, …)

- Distributed Memory
- Message Passing Interface
- Partition

Clusters (MosiX, …)

- COTS networks
- Distribute Linux
- Balance Load dynamically

SMP: Shared Memory / Symmetric MP

- Characteristics of SMP Systems:
  - Highly optimised interconnect networks
  - Shared memory (with several levels of caches)
  - Sizes: 2 .. ~1024 CPUs

- Successful Applications:
  - Large Linux (Windows) machines / servers
  - Transaction-management systems

- Not usually used for:
  - CPU intensive computation, massively parallel Applications

MPP: Massively Parallel Multiprocessors

- Characteristics of MPP Systems:
  - Highly optimised interconnect networks
  - Distributed memory
  - Size today: up to few 100000 CPUs (cores) + XXL GPU
- Successful Applications:
  - CPU intensive computation, massively parallel Applications, small execution/communication ratios
  - Cloud ?
- Not optimal for:
  - Transaction-management systems
  - Unix-Workstation + Servers

“Clusters”

- Characteristics of Cluster Systems:
  - Use COTS (common off the shelf) PCs/Servers and COTS networks
  - Size: No principle limits

- Successful Applications:
  - CPU intensive computation, massively parallel Applications, larger execution/communication ratios
  - Data Centers, google apps
  - Cloud

- Not optimal for:
  - Transaction-management systems
  - Unix-Workstation + Servers

The Limitation of CC

Example:
a numerical application that computes what happens during car crash.

Such simulations typically compute one time step, require some communications about the boundaries and some global variables, and then next time step and so on.

If you compute bus crash, the problem is fairly big, so each time step takes a lot of time - e.g. 1 minute. Even if you use 600 computers efficiently, you'll have 0.1 second per time step, which is a usually enough in terms of communications. So this is coarse-grain.

Same simulations, you check what happens when a hammer left in space impacts a space ship shield. This time the problem is very small, but the velocities and the materials are higher, so the time step is smaller (the physical time) and you need 1,000,000 time steps. Each time step may take 10msec. Impossible to parallelize efficient even on a 100 nodes CC since communication cost is large. This is fine grain, and you'll have to wait A LOT until it finishes.

Oren Laadan/Hermann Härtig (199x)

Parallel Programming Models

- Organisation of Work
  - Independent, unstructured processes (normally executing different programs) independently on nodes (make and compilers, ...), “pile of work”
  - SPMD: single program on multiple data
    asynchronous handling of partitioned data

- Communication
  - Shared Memory, shared file system
  - Message Passing:
    Process cooperation through explicit message passing

- Task queues
- Amnon's slides
- Map/reduce as additional paradigm
- Evt central dispatch

Programming Model: SPMD

- Floyd: SISD, SIMD, MIMD, (MISD)

  SIMD: Single Instruction Multiple Data
  Vector Computers, specialized instructions, ...
- SPMD: Single Program Multiple Data

  Same program runs on “all” nodes
  works on split-up data
  asynchronously but with explicit synch points

  implementations: message passing/shared memory/...

  paradigms:
  “map/reduce” (google) / GCD (apple) / task queues / ...

SPMD continued

- SPMD often:

```
while (true) {
  work
  exchange data (barrier)
}
```

- Common for many MPP:
  "All" participating CPUs: active / inactive

- "All" techniques:
  - Partitioning (HW)
  - Gang Scheduling

- Problem to solve for all variants: Load Balancing (→ later)

Gang Scheduling

- The OS schedules all members of a group of processes at the same time
- Using: priorities (mostly), time-driven scheduling
- Why?

Amdahl's Law and the importance of low-jitter computation and <u>low-latency communication</u>

Compute; communicate; compute; ...

Examples (idealized, take with grain of salt !!!):

- Compute: 10 micro, 100 micro, 1 ms
- Communicate: 5 micro, 10 micro, 100 micro, 1ms
  assuming here: communication cannot be sped up

Interpretation of Amdahl's law for parallel systems:

- ◦ P: section that can be parallelized
- ◦ 1-P: serial section
- ◦ N: number of CPUs

$$\text{Speedup}(P,N) = \frac{1}{\left(1 - P + \frac{P}{N}\right)}$$

Amdahl and communication

Compute(= parallel section),
communicate(= serial section):
possible speedup for N=∞

• 1ms, 100 µs:	1/0.1	→ 10
• 1ms, 1 µs:	1/0.001	→ 1000
• 10 µs, 1 µs:	0.01/0.001	→ 10
• ...		

Amdahl and compute jitter

Jitter:

- Ocassional addition to computation time in one or more processes
- Holds up all other processes

Compute(= parallel section),
jitter (→ add to serial section),
communicate(= serial section):
possible speedup for N=∞

1ms, 100µs, 100 µs:	1/0.2	→ 5
1ms, 100µs, 1 µs:	1/0.101	→ 10
10 µs, 10µs, 1 µs:	0.01/0.011	→ 100
...		

Sources of computation jitter

- Hardware ???
- Application:
  computation imbalance → load balancing (by hand, dynamic)
- Operating systems/libraries/... :
  - Context switch times (gang scheduling)
  - Mutual exclusion
  - „noise“: uncontrolled side activities
  - ...
  -
- → HPC should run micro kernels

OS-jitter consequences

- Keep critical sections as „small“ as possible
- Specialize OS functionality for application (e.g. MPI)
- Common: Remove OS from the critical path:
  - Process communication without OS (DMA)
  - Application isolation without OS (special networks)
  - Broadcast/combination/reduction (→ next slides) thru special network

MPI, very brief overview

- Library for message-oriented parallel programming.
- Programming-model:
  - MPI program is started on all processors
  - Static allocation of processes to CPUs .
  - Processes have “Rank”: 0 ... N-1
  - Each process can obtain its Rank (MPI_Comm_rank).
- Typed messages
- Communicator: collection of processes that can communicate, e.g., MPI_COMM_WORLD
- MPI_Spawn (MPI – 2)
  - Dynamically create and spread processes

MPI - Operation

- Init / Finalize

- MPI-Comm-Rank delivers “rank” of calling process, for example

  MPI_Comm_Rank (MPI_COMM_WORLD, &my-rank)

  ```
     if (my_rank != 0 )
     ...
     else ....
  ```

- MPI_barrier(comm) blocks until all processes called it
- MPI_Comm_Size how many processes in comm

MPI – Operations Send, RCV

```
• MPI_Send (
        void* message,
        int count,
        MPI-Datatype,
        int dest,                /*rank of destination process, in */
        int tag,
        MPI_Comm comm)           /* communicator*/
• MPI_RCV(
        void* message,
        int count,
        MPI-Datatype,
        int src,                 /* rank of source process, in */
                                 /* can be MPI_ANY-SRC */
        int tag,                 /* can be MPI_ANY_TAG */
        MPI_Comm comm,           /* communicator*/
        MPI_Status*   status);   /* source, tag, error*/
```

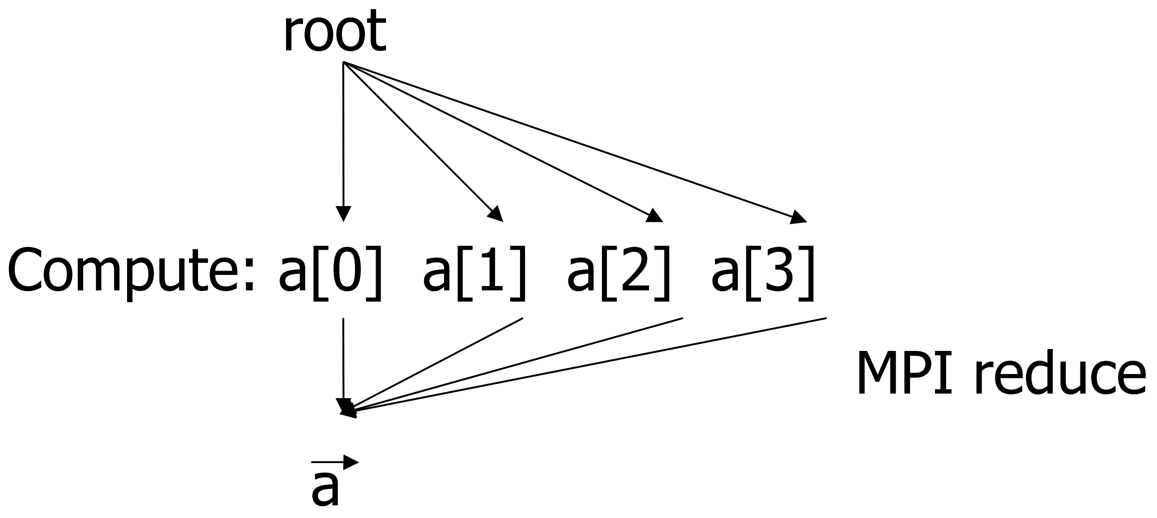
MPI – Operations Broadcast

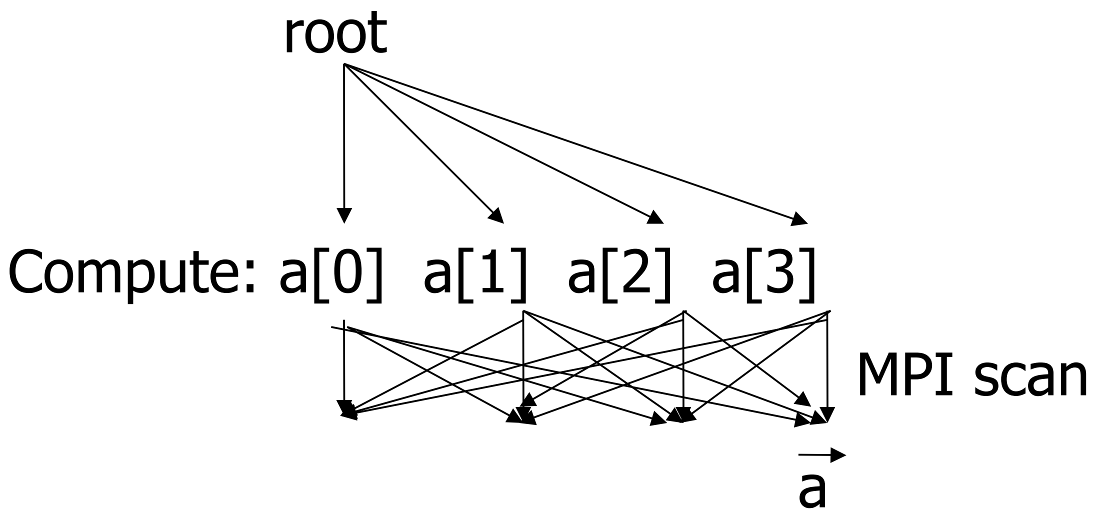
- MPI_BCAST(
  void * message,
  int count,
  MPI-Datatype,
  int root,
  MPI_Comm comm)
- process with rank == root sends,
  all others receive message
- implementation optimised for particular interconnect

MPI – Operations

- Aggregation:
  - MPI_Reduce
    - Each process holds partial value,
    - All processes reduce partial values to final result
    - Store result in RcvAddress field of <u>Root</u> process
  - MPI_Scan
    - Combine partial results into n final results and store them in RcvAddress of <u>all</u> n processes

MPI - Operations





MPI – Operations

- MPI_Reduce(

```
        void* operand,   /* in*/
        void * result,   /* out*/
        int count,       /* in */
        MP_Datatype datatype,
        MPI_Op operator,
        int root,
        MPI_Comm comm)
```

  predefined MPI_OPs:
  sum, product, minimum, maximum,
  logical ops, ...

Common MPP Operating-System-Model
(for example Blue Gene)

- Processing elements/nodes:
  compute intensive part of application
  - Micro-Kernel or stripped down Linux
  - Start + Synchronisation of Application
  - elementary Memory Management (no demand paging)

- all other OS functionality on separate servers or dedicated nodes (“I/O nodes“)

- strict space sharing:
  only one application active per partition at a time

“Space” Allocation in MPP

- Assign partition of field of PEs
  - Applications are pair wise isolated
  - Applications self responsible for PEs
  - shared segments for processes within partition (Cray)

- Problems:
  - debugging (relatively long stop-times)
  - Long-running jobs block shorter jobs

- Isolation of application with respect to:
  - Security
  - Efficiency

“Space” Allocation in MPP

- Hardware-Supported assignment of nodes to applications
- Partitions
  - static at configuration
    Installed by operator for longer period of time
  - Variable(Blue Gene/L):
    Selections and setup on start of Job
    established by “scheduler”
  - Very flexible (not in any MPP I know):
    - increase and shrink during operation
    - Applications need to deal with varying CPU numbers

Alternative: Distribution of Load

- Static
  - Place processes at startup, don’t reassign
  - Requires a priori knowledge
- Dynamic Balancing
  - Process-Migration
  - Adapts dynamically to changing loads
- Problems
  - Determination of current load
  - Distribution algorithm
  - Oscillation possible
- successful in SMPs and clusters, not (yet ?) used in MPPs
- Most advanced dynamic load balancing: MosiX

Challenges for Load Balacing in Clusters (++)

View provided for users/programming model

How to distribute load,

- The mechanism to migrate load
- The mechanisms to use remote resources
- Optimal placement (an NP-Hard problem)
- ...

Information distribution, acting on partial knowledge

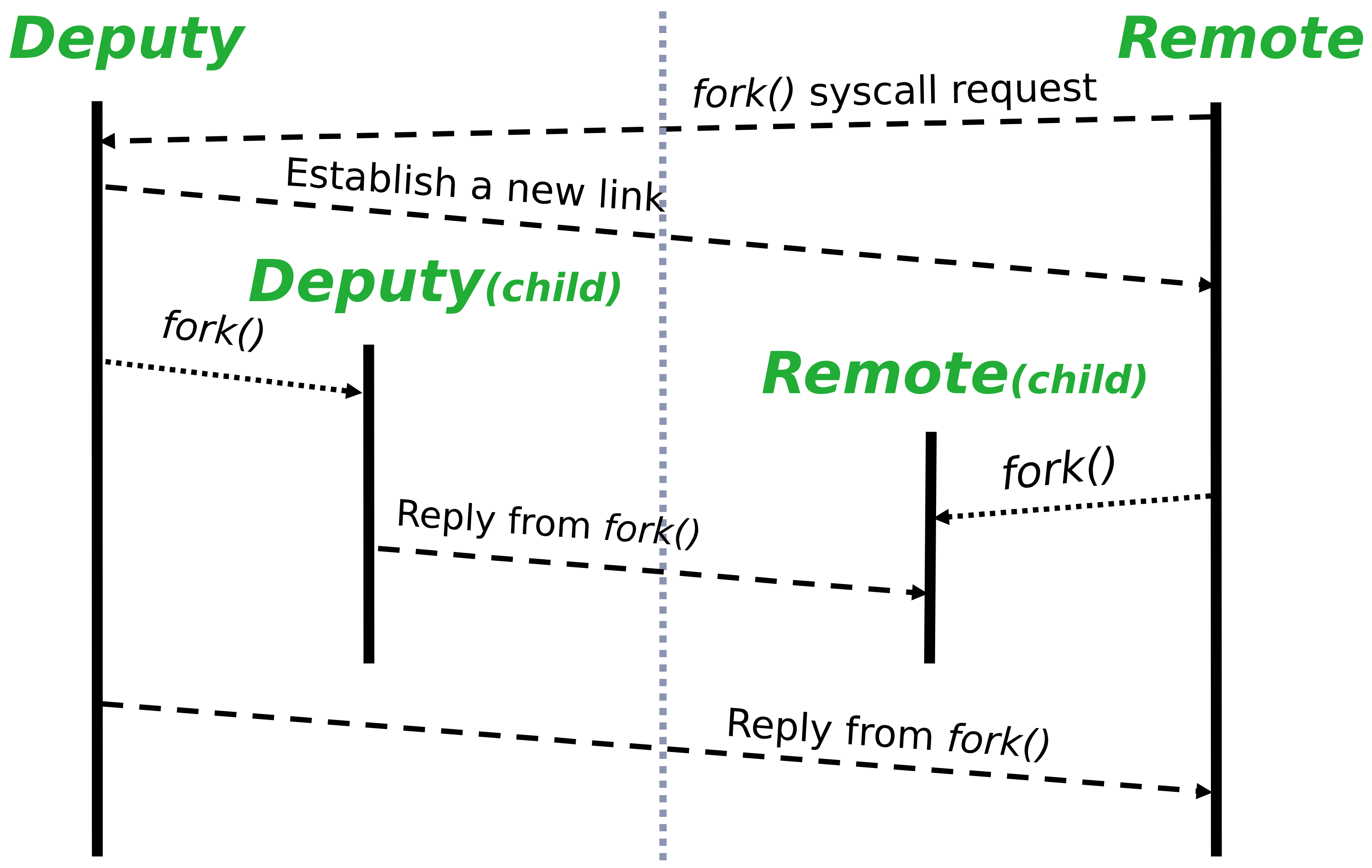
Cope with addition of nodes, subclusters, ...

Administration

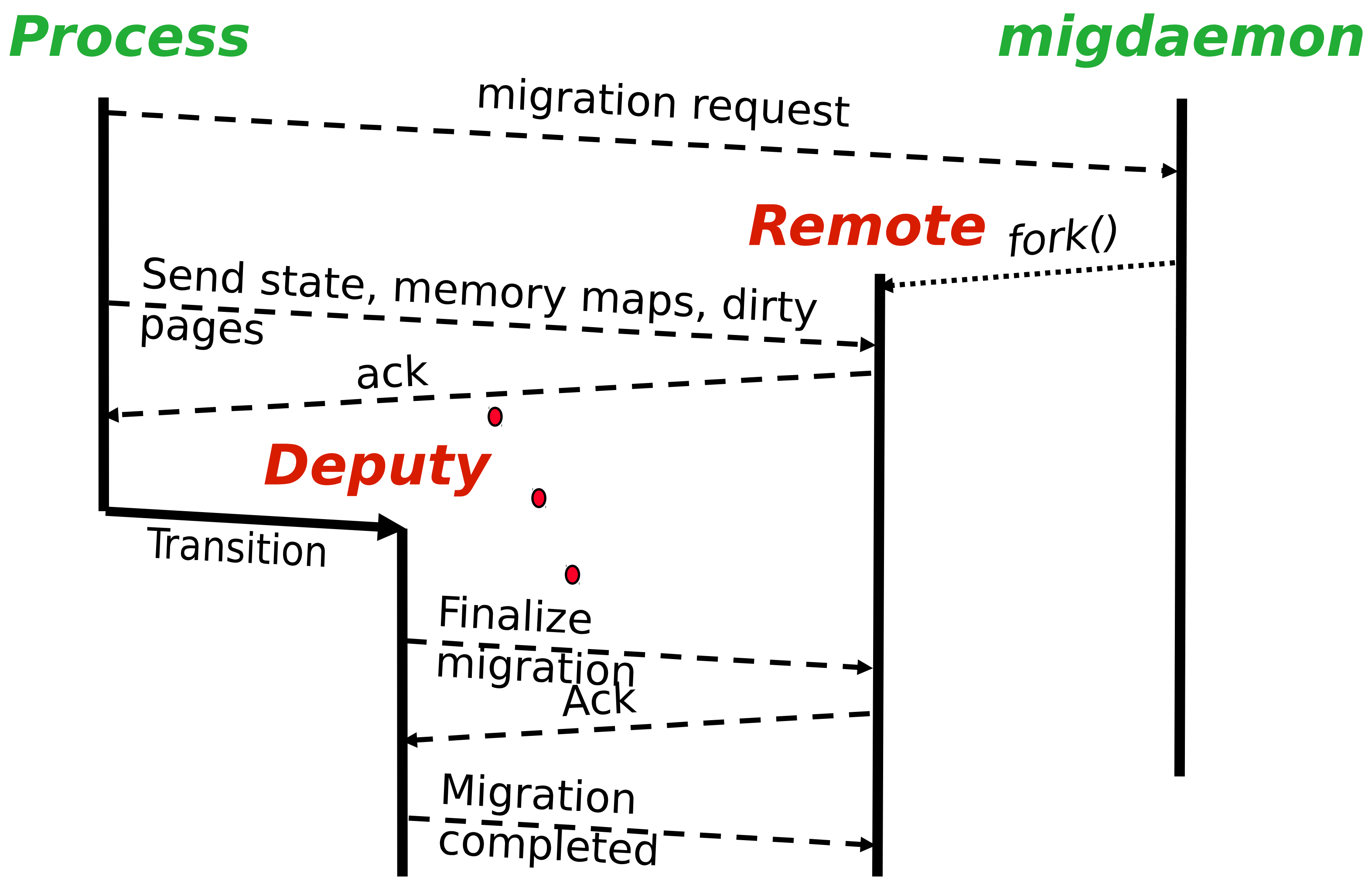
Lots of practical details

NOW: Amnon Barak's slides on MosiX

Special Case: fork()



Process Migration



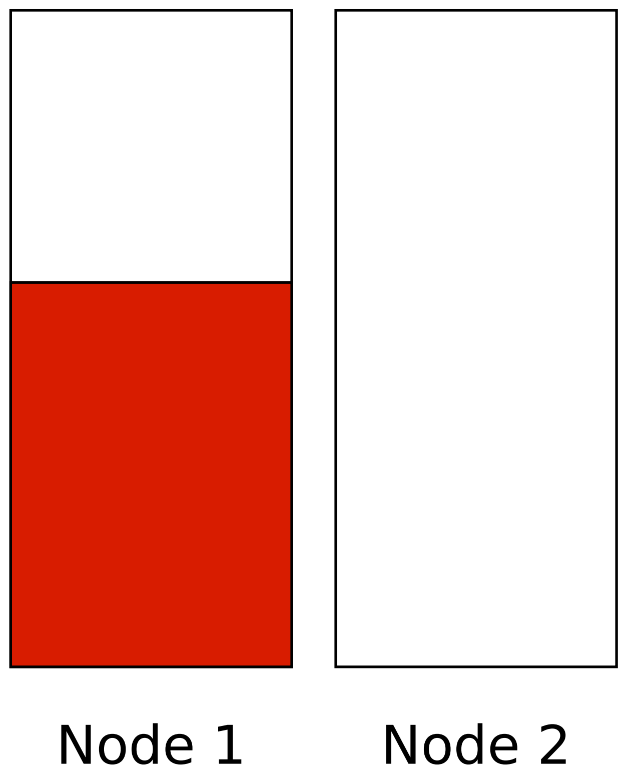
Ping Pong and Flooding

Prevent

- flooding (all processes jump to one new empty node):
  decide immediately before migration commitment (extra communication, piggy packed)

- ping pong:
  if thresholds are very close, processes moved back and forth
  => tell a little higher load than real

The Ping Pong Problem



One process two nodes

Scenario:

- compare load on nodes 1 and 2

  node 1 moves process to equal. loads

  ...

<u>Solutions:</u>

- add one + little bit to load
- average over time

Solves short peaks problem as well

(short cron processes)

The Flooding Problem

Scenario 1: new node comes in

Scenario 2: node becomes unloaded suddenly

=> “everybody joins the party”

Solution:

- use expected load (committed load) instead of run queue length
- check again before committing

IPC

- IPC and load are contradictive
  optimum: NP hard

- apply heuristics: exchange locally