Process Migration – A Review of Dynamic Load-Balancing Algorithms (Part 1)
Hessah Alsaaran

Part 1 Outline
+ Introduction
+ Dynamic Load-Balancing Systems:
  1. MOSIX
  2. Sprite
  3. Mach
  4. LSF
  5. RoC-LB
+ Discussion and Comparison

Part 2 Outline:
• Migration in cloud settings.
• Live migration.
• Any suggested topic...

Definition
+ "The act of transferring a process between two machines during its execution." [1]

Goals (Why use Migration?)
+ Dynamic load distribution.
+ Data locality.
+ Failure recovery (with checkpointing).
+ Administrative shutdowns.

General Algorithm for Migrating Process X

1. Migration request
2. Migration request
3. Migration request
4. Migration request
5. Migration request
6. Migration request
7. Migration request
8. Migration request
9. Migration request
10. Migration request
11. Migration request

Figure 1: High-Level View of Process Migration
Migration → Load Balancing

+ Load information
+ Information about the node and its load.
+ Trade-off between size and network overhead.
+ Load information dissemination:
  + Periodically or event driven.
  + Trade-off between frequency and stability.
+ Load Management and Migration Decisions
  + Centralized or distributed.

Distributed Scheduling

+ The migration decision is distributed.
+ Policies:
  + Sender-initiated policy: the overloaded node requests the migration.
    + Suitable for lightly loaded systems
  + Receiver-initiated policy: the under loaded node requests the migration.
    + Suitable for lightly loaded systems
  + Symmetric policy: combination of both of the above policies.
  + Random policy: randomly choose a node.
    + Simple but improves performance. It can eliminate the management overhead, but it cannot choose the critical nodes that are overloaded or under loaded. (Performance will depend on the load distribution.)

Transfer Strategies

+ Eager (all) strategy: Copy all address space at migration time.
+ Eager (dirty) strategy: Transfer the dirty pages at migration time, then flush when needed.
+ Copy-on-reference strategy: Transfer pages on demand. If it is dirty then it comes from the source node. Otherwise, it can be from the source node or the backing store.
+ Flushing strategy: same as above but the dirty pages are flushed to the backing store.
+ Precopy strategy: The process suspension at the source node is delayed until transfer is complete. Dirty pages may get recopied multiple times.
+ Trade-off: initial migration cost, run-time cost, home-dependency, and reliability.

Alternatives

+ Remote Execution
  + Remote invocation of existing code, or transfer code only.
  + Faster than process migration (no state transfer).
  + But it doesn’t move.
    + Once the process is created at a node, it stays there until it terminates or gets killed.
    + A process cannot resume execution; it is killed then restarted on the other node. (So is it really an alternative?)
  + Can be suitable for very short processes that are not worth transferring their state.

Alternatives

+ Process Cloning
  + Remote forking: children inherit state from their parent.
  + Process migration is resembled by terminating the parent.
  + It involves the process knowledge and action.
  + No resumption here too.

Alternatives

+ Application-level Checkpointing
  + The application checkpoints itself.
  + The application can run using these checkpoints.
  + This is one way of doing process migration, not an alternative.
Alternatives

- Mobile agents
  - Processes that save their own state, move to a new node, and resume execution (coded). (Self-migrating processes)
  - They perform the migration decision.
  - No transparency.

MOSIX - Distributed OS

- Main Feature: Single System Image

Other MOSIX Main Features

- Fully-decentralized control, no global state \(\rightarrow\) scalable.
- Guest (migrated) processes run in sandboxes, and have lower priorities.

2 Types of Processes

- Linux Processes (not affected by MOSIX)
  - Administrative tasks and non-migratable processes.
- MOSIX processes
  - Migratable processes that are created using the "mosrun" command.
  - Each has a home-node: the node where the process was created.

Automatic Resource Discovery

- Randomized gossip algorithm.
  - Each node periodically sends its information along with received information to randomly chosen nodes.
  - Information:
    - Includes: current load, CPU speed, and memory.
    - Load estimated by the average length of the ready queue over a fixed time period. (Is that enough?)
    - Information about new nodes are gradually distributed, while information about disconnected nodes age and get quickly removed.
  - Scalable.
Process Profiling

- Each process has an execution profile:
  - Includes: age, size, rates of system-calls, I/O, and IPC.
  - Used along with load info. for automatic migration decisions.

Dynamic Load-Balancing

- Using process migration.
  - Triggered either manually or automatically.
- Migration: copying process memory image (compressed when sent over the network) and setting its run-time environment (sandbox).
- MOSIX uses the eager (dirty) transfer strategy:
  - At migration time: transfer only the dirty pages of the process.
  - When process resumes execution: text and clean pages are faulted in as needed.

Migration Algorithm

- The migration decision is done:
  - When a node finds another node with significantly reduced load.
  - To migrate processes that request more memory than available.
- The selection for a process to migrate preferentially selects:
  - Older processes.
  - Processes with communication history with that node.
  - Processes with forking history.

Results of the Virtualization Layer

- Migrated processes seem to be running in their home-nodes unaware about the migration.
- Users do not need to know where their programs are running.
- Users do not need modify their applications.
- Users do not need to login or copy files to remote nodes.
- Users have the impression they are running on a single machine.
- A migrated process runs in a sandbox for host protection.

MOSIX’s Virtualization Layer

- MOSIX is a software layer.
- It intercepts all system calls
  - If it is a migrated process: forward it to its home-node, perform the operation on behalf of the process, then send results back.

Drawbacks of the Virtualization Layer

- Migrated processes management and network overhead.
- Table 1 from [1]

Experiment settings:
- 4 identical (Xeon 3.06GHz) servers, with 1Gb/s Ethernet
- 4 applications:
  - RC (satisfiability): CPU-intensive
  - SW (protein sequences): small amount of I/O
  - JELlium (molecular dynamics): large amount of I/O
  - BLAT (bioinformatics): moderate amount of I/O
Migratable Sockets (Direct Communication)

**Without Migratable Sockets**
- Each process has a mailbox.
- A process is identified by its home-node and PID in their home-node. (mapping?)
- Process location is transparent.
- Network overhead is reduced.

**Migratable Sockets**
- Each process has a mailbox.
- A process is identified by its home-node and PID in their home-node. (mapping?)
- Process location is transparent.
- Network overhead is reduced.

Limitations
- Suitable for applications that are compute intensive or have low to moderate I/O.
- Otherwise, network and management overhead.
- For Linux: MOSIX supports non-threaded Linux applications.
- For private clusters: only trusted and authorized nodes.
- For clusters with fast networks.
- Not suitable for the node shutdown case.

Issues Not Discussed
- Node failure: either home-node or remote-node.
- Checkpointing is supported, but where are the checkpoints (saved process image) stored?
- Fully-decentralized? Job queuing system for flood control.
- MOSIX has a job queue that gradually distributes the jobs, according to their priorities, requirements, and currently available resources.
- Where is the job queue?

Sprite
- Like MOSIX:
  - It achieves transparency through the use of home-nodes (single image system).
  - It uses the dirty transfer strategy.
- Unlike MOSIX:
  - Its goal is to utilize idle nodes.
  - Uses a network file system → reduces home-node service.
  - It uses a centralized load information management system.
  - A single process is maintaining the system's state.
  - Scalability issue → bottleneck.

(2) Sprite

(3) Mach
Mach

+ Task migration is a feature added to the Mach microkernel.
+ The process stays on the source node.
+ Only the task's state is transferred.
+ Most system calls are forwarded to the source.

Migration Decision

+ Distributed file system.
+ Distributed scheduling decisions of applications (profiled):
  + Migration to a node for data locality:
    + Significant amount of remote paging.
    + Significant amount of communication.
  + Migration to a node for load balancing:
    + CPU intensive applications.

LSF (Load Sharing Facility)

+ LSF achieves load balancing through:
  + Initial placements of processes (primary goal).
  + Process migration with checkpointing (using Condor's checkpointing).
  + Checkpoints, kills, then restart the process on destination node.
+ Centralized load information management:
  + A single master node.
  + If it fails, another node promotes itself (how?).
  + Other nodes send their information to the master periodically.
  + The master is responsible of scheduling.
  + Based on load information and processes requirements.

Rate of Change Load Balancing (RoC-LB)

+ The idea: the migration decision should depend on how the load changes in time:
  + To predict future overload or starvation.
  + To minimize node idle time.
Each node periodically computes the change in its load since the last time interval to difference in load (DL).

Assuming that the DL will remain the same, each node computes how many time intervals it needs to become idle.

The network delay (ND) is the time it takes to receive a load after a request (adaptive).

If this time less than the network delay (ND) initiate migration request.

Assuming that the DL will remain the same, each node computes how many time intervals it needs to become idle.

If this time less than the network delay (ND) initiate migration request.

If the node's load < CT initiate load request.

Reason: small change in load can cause immediate node starvation.

If the node's load > HT never initiate load request.

Even if it predicted that it will become idle.

Reason:
DL must be high the value is probably rare and short lived.
To resist high spikes in load delay load requests until the load falls below HT.

Each node maintains 2 tables: Sink Table and Source Table.
For every load request it receives, the requesting node is considered a sink.
For every load request reply it receives, the replying node is considered a source.

When a node initiates a load request:
It selects the first source node in the Source Table.
If table is empty select a random node (not in Sink Table).
Send the load request to it.
It may forward the request to another selected source node.

Using DL, the node predicts what its load will be after ND time length predicted load (PL).

If PL ≥ zero the node does not predict idleness in the next ND time period.
If PL < zero the node requests abs(PL) load.

The request receiver will send load:
If its load > HT.
Amount = min (requested amount, a).

Old tasks (CPU time)
Reason: they will probably live longer, making the migration process worthwhile.
Message Passing System and Table Update Algorithm

Load Request Message

<table>
<thead>
<tr>
<th>Sink Process ID</th>
<th>Number of Load Units Requested</th>
<th>Load State of Preceding Nodes</th>
<th>Number of Forwards</th>
</tr>
</thead>
</table>

- **Load Request Sending:** (Requesting Node)
  1. The requesting node will create a message.
  2. Pop a source node from its Source Table.
  3. Send the request message to that source node.

- **Receiving a Reply:** (Requesting Node)
  1. Add the replier to the Source Table.

Receiving Load Request: (Receiving Node)

1. Add the requester node to the Sink Table.
2. If it is a forwarded message, add the preceding node to the table.
3. Check its load.
   - If it can completely fulfill the request:
     1. Send a reply to the requesting node with the requested load.
   - If it can partially fulfill the request:
     1. Send a reply to the requesting node with part of the requested load.
     2. Update the message by:
        - Decreasing the amount of requested load.
        - Incrementing the number of forwards.
        - Change the preceding node field to itself.
     3. Check the number of forwards.
        - If it reached a max value:
          - Discard message and inform the requesting node.
        - Otherwise, forward the message to a source node from the table.
   - If it cannot fulfill the request:
     1. Repeat steps b and c (without changing the amount of requested load).

Comments/Critique

- If an algorithm is balancing the load through "equalization", it covers the case of not wanting idle nodes.
- The requested load only covers the next time interval, then it will need to request another load.
- Repeating process migration process continually (including message passing).
- Trade-off between time interval size and the incurred overhead.
- The node should not only have "enough" load, it should have a mixture of tasks (e.g., CPU-bound and I/O bound) to effectively utilize the resources.

Comparisons & Discussion

Comparisons & Discussion (1)

- Balancing Goal:
  - MOSIX, Mach, LSF \(\rightarrow\) full load balancing
  - RoC-LB, and Sprite \(\rightarrow\) utilizing idle nodes.

Comparisons & Discussion (2)

- Full vs. Partial Process Migration
  - MOSIX, Sprite, Mach \(\rightarrow\) partial migration
  - RoC-LB, LSF \(\rightarrow\) full migration

- In partial migration: (Dependency between the source and destination nodes.)
  - Increased transparency.
  - Management and network overhead. (Inefficiency)
  - Compromises fault-tolerance of the system. (Inefficiency)

- In full migration:
  - Node compatibility becomes an issue.
  - More information needs to be disseminated. (Inefficiency)
  - Longer migration time, but less run-time overhead.
Comparisons & Discussion (3)

- Distributed vs. Centralized load balancing algorithm:
  - MOSIX, Mach, RoC-LB → Distributed.
  - Sprite, LSF → Centralized.
- Distributed algorithms:
  - More scalable.
  - Load information storage issue. (Inefficiency)
- Centralized algorithms:
  - Bottleneck. (Inefficiency)
  - Single point of failure. (Inefficiency)
  - Fast, up-to-date information about all nodes.

Inefficiency → Load Representation

- A node's load is estimated based on the number of load units.
  - Examples:
    - MOSIX estimates load by the average length of the ready queue over a fixed time period.
    - RoC-LB uses load change over time.
- Problem:
  - Load units (processes) are not equal:
    - Different sizes (running times) → Many short tasks can be equivalent to a long one.
    - CPU-bound vs. I/O-bound processes → must have a mixture to effectively utilize the resources.

If I would build a system, how would I do it?

(1) Load balancing through full process migration (no home-node dependency) → more failure tolerant.
- Checkpoints at application level → not transparent but produces significantly smaller checkpoint files that are also machine-independent.
- Replicate these checkpoints at a configurable level on different nodes.
- Periodically save checkpoints and update replicas. If a node holding a replica does not respond, choose a replacement.
- On the other hand, if a timeout occurred at these nodes, one will take over the process.
- When a process has terminated, the node informs the nodes holding replicas to delete these replicas.

(2) Migration decision is distributed as in MOSIX → more scalable.
- But change the load representation to instead of basing on only the number of units as if they are equal:
  - Load units will be weighted by their expected running time.
  - However, this estimation can be an issue.
  - May use history if it has been executed recently on the node.
  - Keeping a configurable ratio of CPU-bound and I/O-bound processes.
  - This information can be extracted from process profiles.
- Instead of sending load info. to 2 random nodes, let the number of random nodes increase with the scale of the system.

(3) Implement the system in the application-level as in LSF → For portability and simplicity, although it may degrade its performance.

Thank You
References


Outline

- "On minimizing the resource consumption of cloud applications using process migrations"
  - by N. Tziritas, S. U. Khana, C.-Z. Xua, T. Loukopoulos, and S. Lalis
- "Rebalancing in a Multi-Cloud Environment"
  - By D. Duplyakin, P. Marshall, K. Keahey, H. Tufo, and A. Alzabarah
  - Science Cloud, 2013.

DRA's Motivation (Problem Statement)

- Cloud services are widely used.
- Pay-as-you-go.
- Minimizing resource consumption is important for users.
- Problem: "Assuming that an application is already hosted in a cloud, reassign the application components to nodes of the system to minimize the total communication and computational resources consumed by the components of the respective application." (Tziritas et al., 2013)
- Using process migration.

Assumptions

1. The nodes are in a tree network structure.
2. The computing requirement of a process does not change dynamically and its execution cost is known a priori.
3. Network model of a cloud environment.

It can be organized in a multi-level hierarchy.
Problem Formulation

Problem: Strategic assignment between processes and nodes (Tziritas et al., 2013). Minimizes application resource consumption, taking capacity into consideration.

Single-Process Migration

The migration is beneficial if the positive load is larger than the negative load.

When does the process stop migrating?

- When it becomes center of gravity:
  - No beneficial migration to any of its 1-hop neighbors.

Possible Problem

- Consider migrating $p_1$ or $p_2$ to $n_2$.

Unbalanced Super-Process Identification

- A super-process that is not at the center of gravity.
Information for Migration Decision

Each node should know:
+ For each process it hosts, the volume of data used in communication both locally and externally.
+ 1-hop communicating neighbors.
+ For each process, the location of its adjacent processes.
+ Concurrent migration of adjacent processes on neighboring nodes is not permitted.

Capacity Constraint

+ Problem: the destination node cannot host a super-process.
+ Solutions in the extended DRA:
  + Prune some processes from the super-process.
  + The least beneficial processes.
  + Enable multi-hop migration.

Inefficiencies and Limitations

+ (1) Local information vs. global information
  + Using only 1-hop migrations can cause a series of migration for a process until it reaches an $x$-hop node.
  + Management and network overhead.
  + Can increase the process execution time. (more in point 4)
+ (2) Load balancing is not considered.
  + Only capacity limit is considered.
  + Communicating processes can overload a node.
  + Can increase the process execution time. (more in point 4)
+ (3) Process length is not considered.
  + Short processes may not be worth migrating.

Inefficiencies and Limitations

+ (4) Application cost = execution cost + communication cost, where the execution cost is fixed.
  + Not true; cost is also affected by how long they use resources (e.g. cost by hour).
  + Migration can cause either speed-up (e.g. if it was placed on an under-loaded node) or slowdown (e.g. continuous migration).
+ (5) Limited by assumptions: tree networks and unchanging process requirements.
  + The authors plan to consider these issues in a future extension.

Rebalancing in a Multi-Cloud Environment

(Motivation (Problem Statement))

+ Multi-cloud environment with user preferences.
+ Example: user uses a community cloud (the preferred cloud), but depending on the availability of its limited resources, the additional work is launched on a public cloud.
+ Automatic rebalancing: downscaling and up-scaling.
+ Automatic up-scaling is provided by Nimbus Phantom and Amazon’s Auto Scaling Service, but automatic downscaling was not addressed.
+ Need to define preferences and a set of policies.
Assumptions

- Multi-clouds are deployed by a central auto-scaling service (authors use an open source Phantom)
- It deploys instances across the clouds and monitors them.
- Master-worker environments
- Distributed worker environments
- Central job queue and scheduler
- The scheduler:
  - Reschedules jobs upon worker failure or rebalancing termination.
  - Provides worker status info. (state, running jobs and their runtime)
- Add and remove worker instances.
- Parallel workflow
- Independent jobs that can be executed without order.

Need for Rebalancing Policies

- Because rebalancing in multi-cloud environment can be disruptive to the workload.
- Terminating an instance can cause the termination of a running job.
  - Job is rescheduled.
  - Increase in total execution time.
- Without rebalancing:
  - Unexpected expenses.

Downscaling Policies

1. Opportunistic-Idle (OI):
   - Wait until there are idle instances in the less desired clouds, then terminate them.
   - This policy continuously terminates excess idle nodes until it reaches the desired state.
2. Force-Offline (FO):
   - Similar to the previous policy, but with graceful termination:
     - Mark as offline, and complete job.

Downscaling Policies

3. Aggressive Policy (AP):
   - Discard partially completed jobs to satisfy request ASAP.
   - To minimize re-execution overhead, choose instances with jobs that have the least running time.
   - Use work threshold for progress.
     - Example: if t<3% and the instance is running a 2-hour job, then the policy can terminate this job only in the first 30 minutes.

Implementation

- Usage of existing technologies:
  - Use infrastructure clouds such as Amazon EC2.
  - The implementation is an integration with the open source Phantom service for auto-scaling.
  - Use HTCondor for a master-worker workload management system.
- Implemented components:
  - The sensor and the rebalancing policy (Python).
  - Communication with the HTCondor masters and Phantom.
Inefficiency

+ Instead of terminating jobs, they can be check-pointed then resumed.
+ Check-pointing is an available service in HTCondor.

Policies Trade-offs

Figure 6: Mean workload execution time (hours), convergence time (hours), workload overhead (percent), and excess cost. Three iterations are run for each policy and no error bar indicates very little deviation. Of and PO also have 0% workload overhead.

References