Part 1: Migration Introduction

• Key reasons: performance and flexibility
• Process migration (aka *strong mobility*)
  – Improved system-wide performance – better utilization of system-wide resources
  – Examples: Condor, DQS
• Code migration (aka *weak mobility*)
  – Shipment of server code to client – filling forms (reduce communication, no need to pre-link stubs with client)
  – Ship parts of client application to server instead of data from server to client (e.g., databases)
  – Improve parallelism – agent-based web searches
Motivation

• Flexibility
  – Dynamic configuration of distributed system
  – Clients don’t need preinstalled software – download on demand

Migration models

• Process = code seg + resource seg + execution seg
• Weak versus strong mobility
  – Weak => transferred program starts from initial state
• Sender-initiated versus receiver-initiated
• Sender-initiated
  – migration initiated by machine where code resides
    • Client sending a query to database server
      – Client should be pre-registered
• Receiver-initiated
  – Migration initiated by machine that receives code
  – Java applets
  – Receiver can be anonymous
Who executes migrated entity?

- Code migration:
  - Execute in a separate process
  - [Applets] Execute in target process
- Process migration
  - Remote cloning
  - Migrate the process

Models for Code Migration

- Mobility mechanism:
  - Weak mobility
    - Sender-initiated mobility
    - Receiver-initiated mobility
  - Strong mobility
    - Sender-initiated mobility
    - Receiver-initiated mobility

- Execution modes:
  - Execute at target process
  - Execute in separate process
  - Migrate process
  - Clone process
Do Resources Migrate?

• Depends on resource to process binding
  – By identifier: specific web site, ftp server
  – By value: Java libraries
  – By type: printers, local devices

• Depends on type of “attachments”
  – Unattached to any node: data files
  – Fastened resources (can be moved only at high cost)
    • Database, web sites
  – Fixed resources
    • Local devices, **communication end points**

Resource Migration Actions

- Actions to be taken with respect to the references to local resources when migrating code to another machine.
- GR: establish global system-wide reference
- MV: move the resources
- CP: copy the resource
- RB: rebind process to locally available resource

<table>
<thead>
<tr>
<th>Process-to-resource binding</th>
<th>Unattached</th>
<th>Fastened</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>By identifier</td>
<td>MV (or GR)</td>
<td>GR (or MV)</td>
<td>GR</td>
</tr>
<tr>
<td>By value</td>
<td>CP ( or MV, GR)</td>
<td>GR (or CP)</td>
<td>GR</td>
</tr>
<tr>
<td>By type</td>
<td>RB (or GR, CP)</td>
<td>RB (or GR, CP)</td>
<td>RB (or GR)</td>
</tr>
</tbody>
</table>

UMassAmherst

Lecture 10, page 7

Lecture 10, page 8
Migration in Heterogeneous Systems

- Systems can be heterogeneous (different architecture, OS)
  - Support only weak mobility: recompile code, no run time information
  - Strong mobility: recompile code segment, transfer execution segment
  - Virtual machines - interpret source (scripts) or intermediate code [Java]

Part 2: Virtual Machine Migration

- VMs can be migrates from one physical machine to another
- Migration can be live - no application downtime
- Iterative copying of memory state
- How are network connections handled?

- Inherently migrates the OS and all its processes
Pre-Copy VM Migration

• 1. Enable dirty page tracking
• 2. Copy all memory pages to destination
• 3. Copy memory pages dirtied during the previous copy again
• 4. Repeat 3rd step until the rest of memory pages is small.
• 5. Stop VM
• 6. Copy the rest of memory pages and non-memory VM states
• 7. Resume VM at destination
• 8. ARP pkt to switch

Post-Copy VM Migration

• 1. Stop VM
• 2. Copy non-memory VM states to destination
• 3. Resume VM at destination
• 4. Copy memory pages on-demand/background
  – Async page fault can be utilized
VM Migration Time

Part 3: Container Migration

- Migration techniques
- Snapshots
- Checkpoint-Resume (CRIU)
Migration Methods

- Cold migration: migrate a VM / container that is shutdown
  - Copy image and data files, start on new machine.
  - No state is preserved

- Warm migration: migrate state from previous instance
  - Suspend running VM/container to disk
  - Copy image, data, suspended memory state
  - Resume execution of suspended VM
  - preserves state, but incurs downtime

- Hot/live migration: migrate state with no downtime
  - Copy state while VM executes; no downtime

Snapshots

- Snapshot: point-in-time copy
  - General concept in operating and distributed systems
  - Snapshots preserve objects (file, disk, VM) as they existed at time of snapshot

- VM Snapshots
  - preserves VM state: memory or disk state
  - Like a backup

- Virtual snapshots: make a virtual copy
  - use copy-on-write to make changes to original

- Snapshots useful for roll-back or migration
  - Snapshots are also known as checkpoints
Checkpoint and Restore

- Warm container migration: Checkpoint and Restore
  - Pause container execution
  - Checkpoint (save) memory contents of container to disk
  - Copy checkpoint to new machine (memory + disk image)
  - Resume execution on new machine

Linux CRIU

- Linux CRIU (Checkpoint Restore In Userspace)
  - Used for warm or live migration, snapshots, debugging
  - Works for individual process and containers migration
- Uses /proc file system to gather all info about each process in the container
  - Save process state (file descriptors, memory state etc)
- Copy saved state to another machine
- CRIU restorer
  - Use fork to recreate processes to be restored
  - Restore resources; for containers, restore namespace
  - TCP repair to restore network sockets on same machine
  - Can migrate active sockets only if IP address moves
    - Use virtual network device in containers and move it
Part 4: Kubernetes (k8s)

- Cluster management using containers
- Container-based Orchestration System
  - Based on Google’s Borg / Omega cluster managers
- Applications are containerized
- K8s will deploy them onto machines of the cluster
  - Replicate app on multiple machines if requested
  - load balance across replicas
  - Can scale up or down dynamically (vary replica pool size, a concept similar to dynamic thread/process pools)
  - Provide automated restart upon detecting failure (self-healing)

K8s Pods

- Pod: contains one or more containers that share volumes and name space
  - Pods: smallest granularity of allocation in k8s.
- Distributed application: multiple components,
  - each component inside a container
  - Each pod consists of one or more components / containers
  - Pod can contain all containers of an application but:
    - If a component needs to be scaled, put each such component in a separate pod
  - Application consists of a set of pods, each independently scalable
    - Pods of an application can span multiple cluster machines

All k8s figures courtesy of
https://www.slideshare.net/rishabhindoria52/introduction-to-kubernetes-139878615
k8s Services

- service: method to access a pod’s exposed interfaces
  - static cluster IP address
  - static DNS name
  - Services are not ephemeral
  - collection of pods
- Pods are ephemeral
  - each has its own IP
  - can be migrated to another machine
  - Pods can communicate with one another using this IP

Control Plane

- **apiserver**: REST interfaces for clients to access management interface
- **etcd**: cluster key-value datastore
  - strongly consistent, highly durable (uses RAFT consensus)
- **controller-manager**: replicate pods, monitor for node failures and restart
- **scheduler**: assigns newly created pods to servers based on resource constraints
- **cloud-controller-manager**: interact with cloud platforms
K8s Node

- **kubelet**: agent on each node
  - ensure containers are running and healthy

- **kubelet proxy**
  - Manage network rules
  - Load balancing for cluster services

- **container runtime**
  - runtime for container execution
  - containerd/docker, cri-o, rkt

---

**Case Study: Viruses and Malware**

- **Viruses and malware are examples of mobile code**
  - Malicious code spreads from one machine to another

- **Sender-initiated**:
  - proactive viruses that look for machines to infect
    - Autonomous code

- **Receiver-initiated**
  - User (receiver) clicks on infected web URL or opens an infected email attachment