UNIBUS: ASPECTS OF HETEROGENEITY AND FAULT TOLERANCE IN CLOUD COMPUTING

Magdalena Slawinska
Jaroslaw Slawinski
Vaidy Sunderam

{magg, jaross, vss}@mathcs.emory.edu
Creating a problem

1. What do I want?
   - Execute an MPI application

2. What do I need?
   - Target resource: MPI cluster
   - FT services: Checkpoint, Heartbeat

3. What do I have?
   - Access to the Rackspace cloud

4. Why might I want FT on cloud?
   - To reduce costs (money, time, energy, …)
   - Reliability
   - …

5. What is the overhead introduced by FT?

6. Can I do that? How?
Problem

Available resource
- Rackspace cloud

User’s requirements
- Execute MPI software
- Target resource: MPI cluster
- Target platform: FT-flavor

User’s resources
- Rackspace cloud (credentials)

Target resource
- EC2 cloud
- Workstations

Resource transformation
- Manually:
  - interaction with web page
  - prepare the image: install required software and dependencies
  - instantiate servers
  - configure passwordless authentication
  - ....
  - 1 man-hour for 16+1 nodes
Unibus: a resource orchestrator

Available resource

Rackspace cloud

User's requirements
- Execute MPI software
- Target resource: MPI cluster
- Target platform: FT-flavor

User's resources
- Rackspace cloud (credentials)

Unibus

Target resource

EC2 cloud

Workstations
Outline

- Unibus – an infrastructure framework that allows to orchestrate resources
  - Resource access virtualization
  - Resource provisioning
- Unibus – FT MPI platform on demand
  - Automatic assembly of an FT MPI-enabled platform
  - Execution of an MPI application on the Unibus-created FT MPI-enabled platform
  - Discussion of the FT overhead
## Unibus resource sharing model

### Table: Comparison of Traditional and Proposed Models

<table>
<thead>
<tr>
<th></th>
<th>Traditional Model</th>
<th>Proposed Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource exposition</td>
<td>Virtual Organization (VO)</td>
<td>Resource provider</td>
</tr>
<tr>
<td>Resource usage</td>
<td>Determined by VO</td>
<td>Determined by a particular resource provider</td>
</tr>
<tr>
<td>Resource virtualization and</td>
<td>Resource providers belonging to VO</td>
<td>Software at the client side</td>
</tr>
<tr>
<td>aggregation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diagram:

- **(a) Traditional Model**
  - Virtual Organization (VO)
  - Resource usage determined by VO
  - Resource virtualization and aggregation

- **(b) Proposed Model**
  - Resource provider
  - Usage determined by a particular resource provider
  - Software at the client side
Handling heterogeneity in Unibus

- Resources exposed in an arbitrary manner as access points
- Capability Model to abstract operations available on provider’s resources
- Mediators to implement the specifics of access points
- Knowledge engine to infer relevant facts

Diagram:
- Unibus
- Capability Model
- Mediators
- Engine
- Network
- Resources
- User
- access points
- access protocols
- implements
- implements
- uses
- access daemon library
Complicating a big picture …

- Resources exposed in an arbitrary manner as access points
- Capability Model to abstract operations on resources
- Mediators to implement the specifics of access points
- Knowledge engine to infer relevant fact
- Resource descriptors to describe resources semantically (OWL-DL)
- Services (standard and third parties), e.g., heartbeat, checkpoint, resource discovery, etc.
- Metaapplications to orchestrate execution of applications on relevant resources

Diagram:

- Unibus access device
- Metaapplications
- Services
- Unibus
- Capability Model
- Mediators
- Services
- Engine
- Resource descriptors
- Network
- Resources
- Access daemon library
- Implements access protocols
- Implements
- User
- Access points
Virtualizing access to resources
Capability Model and mediators

- Capability Model
  - Provides virtually homogenized access to heterogeneous resources
  - Specifies abstract operations, grouped in interfaces
  - Interface hierarchy not appropriate (e.g. fs:ITransferable and ssh:ISftp)

- Mediators
  - Implement resource access point protocols
Virtualizing access to resources

Ssh Mediator

invoke_shell
exec_command
get_subsystem
get_subsytemsftp
....

compatibleWith

sshd

Workstation

ISsh

shell

exec

subsystem
Knowledge engine

User

Mediator’s Developer

Request interface ISsh

Resource: emily

Knowledge Engine (inferring)

Ssh Mediator

- invoke_shell
- exec_command
- get_subsystem

... implements...

ISsh

- shell
- exec
- subsystem

Operating System

- Linux
- ...

Resource

- emily
- ...

Access Point

- Open SshD
- ...

Some compatibleWith...

Knowledge Set

some compatibleWith...

... hasOS...

... hasAccessPoint...

... hasOperation...
Composite operations

- Rs_addhosts dependsOn create_server
- Create_server is implemented by RS Mediator
- Rs_addhosts implements addhosts
- So RS mediator implements addhosts

Composite operations
- Dynamically expand mediator’s operations
- May result in classification of mediators and compatible resources to new interfaces

**Diagram:**
- ISimpleCloud
  - addhosts
  - deletehosts
- IRackspace
  - create_server
  - delete_server
- Composite operation
- rs_addhosts
- implementation
  - dependsOn
  - rs_addhosts
  - a.k.a. addhosts
- create_server
- delete_server
- RS Mediator
- ISimpleCloud_RS.py
Resource access unification via composite operations

Unified interface

- ISimpleCloud
  - addhosts
  - deletehosts

- IRackspace
  - create_server
  - delete_server

Composite operations

- EC2 Mediator
  - run_instance
  - ec2_addhosts

- Rackspace Mediator
  - create_server
  - delete_server
  - rs_addhosts

Different resources, yet semantically similar

Eliminating need of standardization

User
Resource provisioning
Homogenizing resource heterogeneity

- Conditioning increases resource specialization levels
  - **Soft conditioning**
    - changes resource **software capabilities**
    - e.g., installing MPI enables execution of MPI apps
  - **Successive conditioning**
    - enhances resource capabilities in terms of available **access points** (may use soft conditioning)
    - e.g., deploying Globus Toolkit makes the resource accessible via Grid protocols
Transforming Rackspace to FT-enabled MPI platform

User’s requirements:
- Execute software: NAS Parallel Benchmarks (NPB)
- Target resource: MPI cluster
- FT services: Heartbeat, Checkpoint

User’s credentials

Rackspace descriptor

Metaapp

Unibus

User

FT MPI cluster

Rackspace

Soft conditioning
- Successive cond.
- Composite ops
- ...

NPB logs
Rackspace Cloud to MPI cluster

Installing other services (FT)

Deployment of MPI on new resources

Creating a new group of resources (Rackspace ssh-enabled servers) in terms of new access points

Obtaining a higher level of abstraction

Legend

- Alternative transformation fromInterface to tInterface
- Applied transformation fromInterface to tInterface
- Resource interface, #resource name
- Alternative resource
User’s requirements
- Execute software: NAS Parallel Benchmarks (NPB)
- Target resource: MPI cluster
- FT services: Heartbeat, Checkpoint
Metaapplication

- Requests
  - IClusterMPI
  - FT services:
    - IHeartbeat
    - ICheckpointRestart
- Specifies available resources
- Performs benchmarks
- Transfers benchmarks execution logs to the head node
- Requests ISftp

```python
def create_cluster(resource):
    proxy = resource.createProxy('IClusterMPI', services=('IHeartbeat', 'ICheckpointRestart'), cpus=4)  # 4 core processors
    # install NPB3.3 and dependencies (gcc, gfortran, OpenMPI)
    # checkpoint every 1 minutes, heartbeat every 10 seconds
    proxy.services['IHeartbeat'].init(interval=10, death_after=2)
    proxy.services['ICheckpointRestart'].init(interval=1+60)

import_resources('file:resources.owl')
# the resource needs to be MPI cluster and have some features
res = get_resource('a cluster:IClusterMPI;
    uc:compatibleWithService uc:ICheckpointRestart, uc:IHeartbeat;
    cluster:cpuCores ?n . filter(?n >= $d) % cpus)
proxy = create_cluster(res)

tests = BenchmarkTestsIterator()
for test in tests:
    test_file = '.'.join(test, class_, cpus)
    cmd = '.'.join(path, test_file, '>', log_path)
    try:
        proxy['#mpiexec'](cmd, cpus)
    except proxy.services['IHeartbeat'][#Exception]():
        # is any chkp files available (to restart)?
        if proxy.services['ICheckpointRestart'][#get]():
            while True:
                new_proxy = create_cluster(res)
                cpkt = proxy.services['ICheckpointRestart'][#get]()
                new_proxy.services['ICheckpointRestart'][#set](cpkt)
                # remove old, failed cluster
                proxy = new_proxy
                try:
                    proxy.services['ICheckpointRestart'][#restart]()
                except proxy.services['IHeartbeat'][#Exception]():
                    continue
                else:
                    break
            # no chkpt available yet — just restart the same test
        else:
            tests.repeat()
    # test done — save the result log locally
    with open('.' + os.path.join((test_file, time.strftime('%y%m%d%H%M%S'))), 'w') as f:
        # get the ISftp interface to the head node
        head = proxy['#get_head_node']().createProxy('ISftp')
        f.write(head['open'](log_path).read())
```

---

**Metaapplication**

- Requests
  - IClusterMPI
  - FT services:
    - IHeartbeat
    - ICheckpointRestart
- Specifies available resources
- Performs benchmarks
- Transfers benchmarks execution logs to the head node
- Requests ISftp
Rackspace testbed

- 16 working nodes (WN) + 1 head node (HN)
- Node: 4-core, 64-bit, AMD 2GHz
- Debian 5.0 (Lenny)
- OpenMPI v. 1.3.4 (GNU suite v. 4.3.2 (gcc, gfortran)
- NAS Parallel Benchmarks v.3.3, class B

FT setup:

Heartbeat service:
- OpenMPI-based – in case of failure, the service determines failed node(s) and raises an exception

Checkpoint/restart service:
- DMTCP – Distributed MultiThreaded Checkpointing
  - user-level transparent checkpointing
- Executes dmtcp_command every 60 secs on HN to checkpoint 81 processes (64 MPI processes, 16+1 OpenMPI supervisor processes)
- Moves local checkpoint files from WN to HN (in parallel)
- Checkpoint time – 5 sec; moving checkpoints from WN -> HN less than 10 sec; compressed checkpoint size c.a.1GB
Results: NPB, class B, Rackspace, DMTCP, OpenMPI Heartbeat

16 Worker Nodes (WN) + 1 Head Node
WN: 4-core, 64-bit, AMD Opteron 2GH, 1GB RAM, 40 GB HDD
Checkpoints every 60 sec, average of 8 series

Checkpoints every 60 sec

FT overhead 2% - 10%

HB - Heartbeat

HB + FT
The Unibus infrastructure framework
- Virtualization of access to various resources
- Automatic resource provisioning

Innovatively used to assemble an FT MPI execution platform on cloud resources
- Reduces effort to bare minimum (servers instantiation, etc)
  - 15-20 min from 1 man-hour
- Observed FT overhead 2%-10% (expected at least 8%)

Future work
- Migration and restart of MPI-based computations on two different clouds or a cloud and a local cluster
- Work with an MPI application