Multi-Agent Negotiation of Virtual Machine Migration Using the Lightweight Coordination Calculus

Paul Anderson, Shahriar Bijani, and Alexandros Vichos

School of Informatics, University of Edinburgh, UK
dcspaul@ed.ac.uk, s.bijani@ed.ac.uk, alexandros.vichos@gmail.com

Abstract. LCC is a Lightweight Coordination Calculus which can be used to provide an executable, declarative specification of an agent interaction model. In this paper, we describe an LCC-based system for specifying the migration behaviour of virtual machines in a datacentre. We present some example models, showing how they can be used to implement different policies for the machine allocation and migration. We then describe a practical implementation of the system which can directly execute the LCC specifications.

Keywords: autonomic computing, multi-agent systems, virtual machines, OpenKnowledge, Lightweight Coordination Calculus

1 Introduction

Virtualisation technology has recently transformed the availability and management of compute resources. Each *physical machine* (PM) in a datacentre is capable of hosting several *virtual machines* (VMs). From the user's point of view, a virtual machine is functionally equivalent to a dedicated physical machine; however, new VMs can be provisioned and decommissioned rapidly without changes to the hardware. VMs can also be *migrated* between physical machines without noticeable interruption to the running applications. This allows dynamic load balancing of the datacentre, and high availability through the migration of VMs off failed machines. The resulting *virtual infrastructure* provides the basis for *cloud computing*.

Managing the placement and migration of VMs in a datacentre is a significant challenge; existing commercial tools are typically based on a central management service which collates performance information from all of the VMs. If the current allocation is unsatisfactory (according to some policies), then the management service will compute a new VM allocation and direct agents on the physical machines to perform the necessary migrations.

As the size and complexity of datacentres increases, this centralised management model appears less attractive; even with a high-availability management service, there is possibility of failure and loading problems. If we would like to extend the domain of the virtual infrastructure to encompass multiple datacentres,

managed by different providers, then the central model is no longer appropriate; in this federated "cloud" scenario, there may no longer be a single organisation with ultimate authority over all of the infrastructure.

This motivates us to propose a less centralised solution where agents located on the physical machines negotiate to transfer VMs between themselves, without reference to any centralised authority. This seems particularly appropriate for many situations where a globally optimal solution is not necessary or feasible; for example, if a machine is overloaded, it is often sufficient to find some other machine which will take some of the load. Likewise, an underloaded machine simply needs to take on additional VMs to improve its utilisation; there is no need for any global knowledge or central control.

In this paper, we present an experimental implementation of the above scenario in which agents follow *interaction models* (IMs) described in the *lightweight coordination calculus* (LCC). The agents use the OpenKnowledge framework to locate appropriate interaction models and to identify suitable peers. These interaction models specify the agent behaviour, and allow them to make autonomous decisions; for example, the choice of VM to accept could be based on local capabilities, the properties of the VM being offered, the financial relationship with the donor, etc.

One important consequence of this approach is that we can very easily change the global policy of an entire infrastructure by introducing new interaction models. For example, a particular model may encourage the physical machines to distribute the load evenly among themselves; this makes a lightly-loaded infrastructure very agile and able to accept new VMs very quickly. Alternately, a different interaction model may encourage the machines to prefer a full, or empty, loading as opposed to a partial one. Some of the machines would then be able to dispose of all their VMs, allowing them to be turned off and hence saving power.

2 LCC and OpenKnowledge

A computational agent - such as one responsible for one of our physical machines - must be capable of acting autonomously, but it will also need to communicate with other agents in order to achieve its goals. In a multi-agent system (MAS), the agents often observe conventions which allow them to co-operate. These are analogous to the *social norms* in human interactions, and may be more or less formal – an oft-cited example is the rules which govern the bidding process in an auction. In our application, agents must be able to compare the respective resource utilisation of their hosts, and reach an agreement about the transfer of a virtual machine. Typically, the social norms in a MAS will be defined using an explicit protocol. The *lightweight coordination calculus* (LCC) is a declarative, executable specification language for such a protocol.

LCC is based on a process algebra which supports formal verification of the interaction models. In contrast with traditional specifications for *electronic in-*

stitutions, there is no requirement to predefine a "global" script which all agents follow - the protocols can be exchanged and evolved dynamically during the conversation. LCC is used to specify "if" and "when" agents communicate; it does not define how the communication takes place[1], and it does not define how the agents rationalise internally. There are several different implementations of the LCC specification, including OpenKnowledge (see below), Li[2], UnrealLCC[3] and Okeilidh[4].

There is insufficient space here to describe the LCC language in detail; the OpenKnowledge website contains a good introduction[5], and there are also some video tutorials[6]. The following brief summary should be sufficient to follow the annotated examples presented in the next section:

Each IM includes one or more clauses, each of which defines a *role*. Each role definition specifies all of the information needed to perform that role. The definition of a role starts with: `a(`*roleName, PeerID)*. The principal operators are outgoing message (`=>`), incoming message (`<=`), conditional (`<-`), sequence (`then`) and committed choice (`or`). Constants start with lower case characters and variables (which are local to a clause) start with upper case characters. LCC terms are similar to Prolog terms, including support for list expressions. Matching of input/output messages is achieved by structure matching, as in Prolog.

The right-hand side of a conditional statement is a *constraint*. Constraints provide the interface between the IM and the internal state of the agent. These would typically be implemented as a Java *component* which may be private to the peer, or a shared component registered with a discovery service.

OpenKnowledge (OK[7])[7,12] provides an implementation of LCC, together with some additional functionality, including a distributed *discovery service*. Peers register their desired roles with the service, and this identifies a suitable set of peers to engage in a particular interaction. The peers are then notified and the interaction proceeds without further involvement of the discovery service[8]. OK is capable of quite sophisticated ontology-based role matching, and negotiation among the peers when attempting to fill the roles for a particular interaction. However, we only make use of the basic matching functions - for example, matching an "overloaded" peer with an "underloaded" one.

The OK discovery service also provides facilities for discovering and distributing both interaction models and components (OKCs). This means that a

[1] The inter-agent communication mechanism is defined by the implementation.
[2] `http://sourceforge.net/projects/lij`
[3] `http://sourceforge.net/projects/unreallcc`
[4] `http://groups.inf.ed.ac.uk/OK/drupal/okeilidh`
[5] `http://groups.inf.ed.ac.uk/OK/index.php?page=tutorial.txt`
[6] `http://stadium.open.ac.uk/stadia/preview.php?whichevent=984&s=29`
[7] `http://groups.inf.ed.ac.uk/OK/`
[8] In practice, one of the peers is elected as a coordinator for the interaction, and the coordinator executes the IM, only making calls to other peers when it is necessary to evaluate a constraint

physical machine (in our application) need only register its willingness to participate, and the behaviour will then be defined by the IMs and OKCs which are retrieved from the discovery service. Of course, individual machines remain free to fulfil their role in whatever way is appropriate - possibly by using their own IMs and/or OKCs rather than those available on the central discovery service. This allows particular machines, or federated groups to engage in the same interactions, using different policies.

3 Interaction Models for VM Migration

In this section, we describe two LCC interaction models, which implement two different VM migration policies:

- in the first policy, VMs migrate from busy peers to underloaded peers to balance the load of each peer (Figure 1).
- in the second policy, VMs migrate from underloaded peers to busy peers to fully use the resources of some peers and release the others.

Figure 1 illustrates the state diagram of the first policy. There are three states: *idle, overloaded* and *underloaded*. The *idle* state is the initial and the goal state, in which the peer is balanced. Each peer is assumed to be balanced at the beginning of the interaction. It may then change state, based on its status. Each overloaded peer interacts with an underloaded peer (if any exist), in order to balance the load.

```
// Definition of the "idle" role. Here, "idle" means the "balanced" state
a(idle, PeerID) ::
   // the constraint to check the state of the peer
   null <- getPeerState(Status) then
   // select the next state based on the peer's status
   ( null <- isOverLoaded() then // if the peer is overloaded,
   // change the peer's role to "overloaded" and pass the status
   a(overloaded(Status), PeerID)
   )   or
   ( null <- isUnderLoaded() then // if the peer is underloaded,
   a(underloaded(Status), PeerID) // change the role to "underloaded"
   )   or
   a(idle, PeerID) // otherwise, remain in the idle role (recursion)

// Definition of the "overloaded" role. "Need" is the amount of resources required
a(overloaded(Need), ID1) ::
   // send the "readyToMigrate(Need)" message to an underloaded peer
   readyToMigrate(Need) => a(underloaded, ID2) then
  // wait to receive "migration(OK)" from the underloaded peer
   migration(OK)<= a(underloaded, ID2) then
   // do the migration: send VMs from this peer to the underloaded peer
   null <- migration(ID1, ID2) then
```

```
   a(idle, ID1) // change the peer's role to "idle"

// Definition of the "underloaded" role: "Capacity" is the amount of free resources
a(underloaded(Capacity), ID2) ::
   // receive the "readyToMigrate(Need)" message from an overloaded peer
   readyToMigrate(Need)<= a(overloaded, ID1) then
   // send back the "migration(OK)" message, if the migration is possible
   migration(OK) => a(overloaded, ID1)
                    <- isMigrationPossible(Capacity, Need) then
   null <- waitForMigration() then
   a(idle, ID2) // change the peer's role to "idle"
```

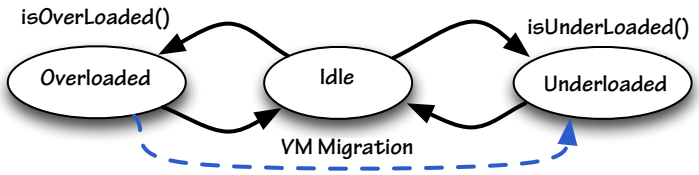

Fig. 1. The state diagram of the first migration policy: unbalanced peers interact to balance their loads.

It may be that we would prefer to have the minimum number of active peers, each using almost all of their resources (e.g. to minimise the cost). This change of policy could be easily deployed by changing only some parts of the IM in the above LCC code. The following IM is an example of the second policy - it is very similar to the first policy, but it has one more state (*shutdown*) for the free peers with no load:

```
// In this policy, "idle" means the fully-loaded state, which is the goal state.
//     i.e the peer uses all available resources
a(idle, PeerID) ::
   null <- getPeerState(Status) then
   // if the peer's load > threshold (e.g. %50 ), but it still has free resources
   ( null <- canAcceptMoreLoad() then
    // change the peer's role to "notFullyLoaded" and send the peer's status
    a(notFullyLoaded(Status), PeerID)
   )   or
   ( null <- isUnderLoaded() then // if the peer is underloaded (e.g.≤ %50)
    a(underloaded(Status), PeerID) // change the role to "underloaded"
   )   or
   ( null <- hasNoLoad() then // if the peer has no load
    a(shutdown, PeerID) // change the role to "shutdown"
   )   or
   a(idle, PeerID) // otherwise, the peer is fully-loaded (recursion)
```

```
// Definition of the "notFullyLoaded" role. "Capacity" = free resources
a(notFullyLoaded(Capacity), ID1) ::
   // send the "readyToMigrate(Capacity)" message to an underloaded peer
   readyToMigrate(Capacity) => a(underloaded, ID2) then
  // wait to receive "migration(OK)" from the underloaded peer
   migration(OK)<= a(underloaded, ID2) then
  // VM migration from an underloaded peer to this peer (notFullyLoaded)
   null <- migration(ID2, ID1) then
   a(idle, ID1) // change the peer's role to "idle"

// Definition of the "underloaded" role. "Load" is the amount of busy resources
a(underloaded(Load), ID2) ::
  // receive the "readyToMigrate(Capacity)" message from a notFullyLoaded peer
   readyToMigrate(Capacity)<= a(notFullyLoaded, ID1) then
   // send back the "migration(OK)" message, if the migration is possible
   migration(OK) => a(overloaded, ID1)
                     <- isMigrationPossible(Capacity, Load) then
   null <- waitForMigration() then
   a(shutdown, ID2) // change the peer's role to "shutdown"

// Definition of the "shutdown" role
a(shutdown,ID3)::
   null <- releaseResources() then // the peer's resources will be released
   null <- sleep(5000) then // wait 5 second
   a(idle, ID3) // after wake up, change the peer's role to "idle"
```

The level granularity of the LCC code in reflecting the details of the policy implementation is optional. Instead of implementing the details of the policy (e.g. delays, etc.) in LCC code we could push them down into the Java constraints.

4 A Prototype

To validate the approach in a realistic environment, we constructed a small prototype, based on a cluster of physical machines (see figure 2). This consisted of four HP Proliant DL120 G6 servers each with 4GB memory, and each capable of supporting in the order of 10 virtual machines. The configuration of this cluster is described fully in [13]. Briefly:

1. We chose to use KVM[9] as the default hypervisor. This is a loadable kernel module that converts the Linux kernel into a bare metal hypervisor. It is a mainstream component of Linux distributions (Fedora and Redhat), it is freely available, well-supported locally, and has a programmable interface via `libvirt`. We would expect the prototype to be equally implementable on top of other common hypervisors such as VMware or Xen.

[9] `http://www.linux-kvm.org/`

2. We used the `libvirt`[10] library to provide a programmable API to the underlying hypervisor (KVM). This supports full remote management of the virtual machines, including stop/start and migration. The KVM C API was exposed to our Java components using JNA.
3. Shared access to the VM images is necessary for migration, and we used an NFS filesystem to store the images.
4. We wrote code to interface OpenKnowledge components (and/or peers) with both `libvirt` and the underlying OS facilities. This provided control of the virtual machines, and information on the state of the system - such as memory and CPU usage.
5. We used the standard OpenKnowledge discovery service to provide peer discovery and distribution of the components and interaction models. A single discovery server was adequate in this case, although OpenKnowledge uses the Pastry overlay network which is capable of supporting a redundant peer-to-peer network of connected discovery servers.

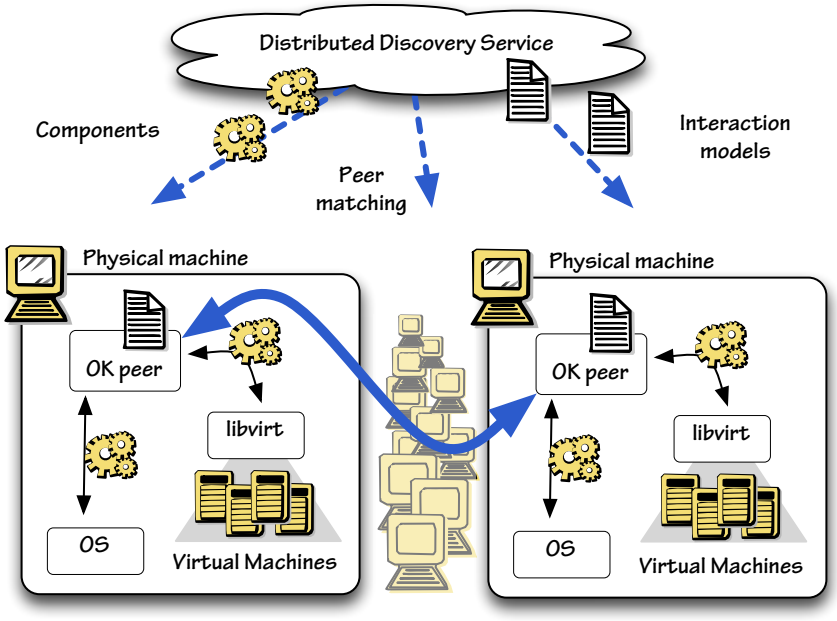

Fig. 2. The architecture of the prototype described in section 4

[10] `http://libvirt.org/`

5 Evaluation

In addition to the experiments with the live prototype, we used a simple simulator to investigate the behaviour of more complex models with a larger number of machines and more controlled loading. Figure 3 shows the results of 50 simulated virtual machines running on 15 physical machines. In this example, physical machines offload VMs if they have a load greater than 120% of the average, and they accept VMs if they have a load less than 80%. Initially, the VMs are allocated randomly and the resulting load is uneven.The system stabilises after a time with all the physical machines except one within the desired range (the load on the remaining machine cannot be reduced because all of the machines have a load greater than 80%). Further results and details of the simulator are available in [5].

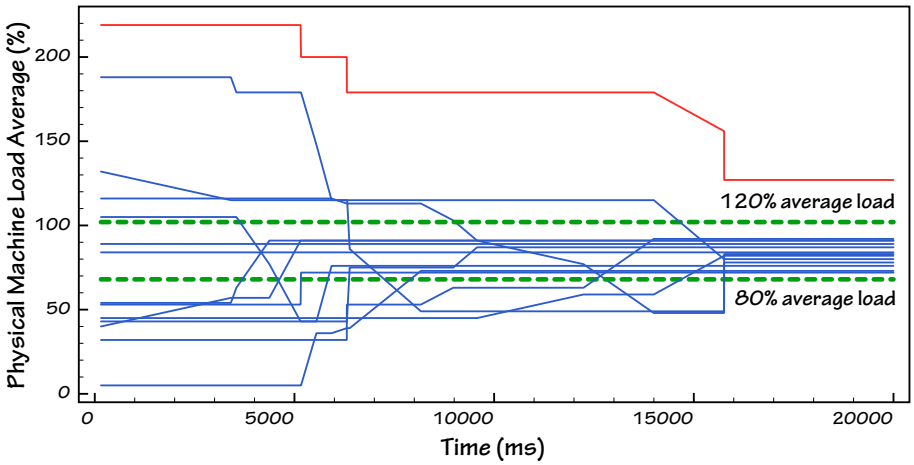

Fig. 3. A simulation showing the load on 15 physical machines as they interact to balance a load of 50 virtual machines.

6 Related Work

There is a considerable amount of existing work on load balancing of virtual infrastructures (see, for example [1,2,15]). Most of this work assumes a central service which collects monitoring data from the physical and virtual machines, computes any necessary re-allocation, and orchestrates the appropriate migrations. However, this leads to difficulties in managing the interactions of imperative control algorithms[10], and limits the degree to which it is possible to exploit the resources of a more federated environment[3,9].

VMWare is a popular provider of commercial management infrastructure for virtual datacentres. The VMWare *vSphere Distributed Resource Scheduler* (DRS) product allows the user to specify rules and policies to prioritise how resources are allocated to virtual machines. DRS[11] "continuously monitors utilisation across resource pools and intelligently aligns resources with business needs" . *vSphere Distributed Power Management* (DPM) allows workloads to be consolidated onto fewer servers so that the rest can be powered-down to reduce power consumption. Citrix Essentials[12] and Virtual Iron "Live capacity" [13] are other commercial products offering similar functionality, and LBVM[14] is an open-source product based on Red Hat Cluster Suite. However, all of these products use a centralised management model.

As noted by Kephart and Walsh[4], agent-based technologies are a natural fit for implementing *autonomic* systems[6]; CatNets[11], for example, is a market-based resource management system. Several people have applied agent-based techniques to virtual machine management: Xing[16] describes a system where "each virtual machine can make its own decision when and where to migrate itself between the physical nodes" - for example, two VMs may notice that the applications running on them are communicating frequently, and the VMs may decide that they should attempt to migrate so that they are physically closer. Spata and Rinaudo[8] describe a FIPA-compliant system with very similar objectives to our own which is intended to load-balance VMs across a cluster. However, we are not aware of any other implementation which is driven directly from a declarative specification of the interaction model.

7 Conclusions and Future Work

We have demonstrated that an agent-based approach using LCC interaction models is a viable technique for negotiating virtual machine placement and migration. This is especially appropriate where the number of machines involved is very large and global knowledge is neither possible, nor necessary. It is also applicable in federated situations where there is no single point of control, or policy.

We have only described comparatively simple interactions, but the abstraction provided by the LCC model makes this an ideal basis to explore more complicated scenarios. These might involve more sophisticated negotiations (such as auctions), and/or more dimensions to the underlying metrics (such as memory usage, bandwidth, proximity, etc.).

---

[11] `http://www.vmware.com/pdf/vmware_drs_wp.pdf`

[12] `https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=HPE4XSE`

[13] `http://www.storageengineers.com/pdf_virtualiron/Evaluation_Guide_0107.pdf`

[14] `http://lbvm.sourceforge.net/`

Acknowledgements: We would like to thank HP Labs Bristol, UK for providing the hardware for the experimental cluster, and Jun Le for allowing us to use the data from his simulations.

References

1. Bobroff, N., Kochut, A., Beaty, K.: Dynamic placement of virtual machines for managing SLA violations. Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on pp. 119–128 (21 2007-Yearly 25 2007)
2. Bodk, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., Patterson, D.: Statistical machine learning makes automatic control practical for internet datacenters. In: Proceedings of Workshop on Hot Topics in Cloud Computing (HotCloud) (2009)
3. Grit, L., Irwin, D., Aydan, Chase, J.: Virtual machine hosting for networked clusters: Building the foundations for "autonomic" orchestration. Virtualization Technology in Distributed Computing, 2006. VTDC 2006. pp. 7–7 (Nov 2006)
4. Kephart, J., Walsh, W.: An artificial intelligence perspective on autonomic computing policies. In: Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on. pp. 3 – 12 (june 2004)
5. Li, J.: Agent-based management of Virtual Machines for Cloud infrastructure. Master's thesis, School of Informatics, University of Edinburgh (2011)
6. Murch, R.: Autonomic Computing. IBM Press, 1 edn. (2004)
7. Pinninck, A.P.D., Kotoulas, S., Siebes, R.: The OpenKnowledge kernel. In: Proceedings of the IX CESSE conference (2007)
8. Rinaudo, M.O.S..S.: Virtual machine migration through an intelligent mobile agents system for a cloud grid. In: Journal of Convergence Information Technology, vol. 6. Advanced Institute of Convergence Information Technology (June 2011)
9. Ruth, P., Rhee, J., Xu, D., Kennell, R., Goasguen, S.: Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. Autonomic Computing, 2006. ICAC '06. IEEE International Conference on pp. 5–14 (June 2006)
10. Schmid, M., Marinescu, D., Kroeger, R.: A Framework for Autonomic Performance Management of Virtual Machine-Based Services. In: Proceedings of the 15th Annual Workshop of the HP Software University Association (June 2008)
11. Schnizler, B., Neumann, D., Veit, D., Reinicke, M., Streitberger, W., Eymann, T., Freitag, F., Chao, I., Chacin, P.: Catnets deliverable 1.1: Theoretical and computational basis. Tech. rep., CatNet Project (2005)
12. Siebes, R., Dupplaw, D., Kotoulas, S., de Pinninck Bas, A.P., van Harmelen, F., Robertson, D.: The OpenKnowledge System: an interaction-centered approach to knowledge sharing. In: Meersman, R., Tari, Z. (eds.) Lecture Notes in Computer Science. vol. 4803, pp. 381–390. Springer, Springer (2007)
13. Vichos, A.: Agent-based management of Virtual Machines for Cloud infrastructure. Master's thesis, School of Informatics, University of Edinburgh (2011)
14. Walton, C., Robertson, D.: Flexible multi-agent protocols. Tech. rep., University of Edinburgh (2002)
15. Wood, T., Shenoy, P., Venkataramani, A., Yousif, M.: Black-box and gray-box strategies for virtual machine migration. In: Proceedings of the 4th Usenix Symposium on Networked Systems Design and Implementation. Usenix (April 2007)
16. Xing, L.: A Self-management Approach to Service Optimization and System Integrity through Multi-agent Systems. Master's thesis, University of Oslo, Department of Informatics (May 2008)