www.ietdl.org

Published in IET Software
Received on 23rd November 2009
Revised on 26th July 2010
doi: 10.1049/iet-sen.2009.0091

ISSN 1751-8806

Using lightweight virtual machines to achieve resource adaptation in middleware

H.A. Duran-Limon[1] *M. Siller*[2] *G.S. Blair*[3] *A. Lopez*[2] *J.F. Lombera-Landa*[2]

[1]*Information Systems Department, CUCEA, University of Guadalajara, Periferico Norte #799, Núcleo Belenes, Zapopan 45100, Jalisco, México*
[2]*CINVESTAV, Unidad Guadalajara, Av. Científica 1145, Col el bajío, Zapopan 45015, Jalisco, México*
[3]*Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK*
E-mail: hduran@cucea.udg.mx

Abstract: Current middleware does not offer enough support to cover the demands of emerging application domains, such as embedded systems or those featuring distributed multimedia services. These kinds of applications often have timeliness constraints and yet are highly susceptible to dynamic and unexpected changes in their environment. There is then a clear need to introduce adaptation in order for these applications to deal with such unpredictable changes. Resource adaptation can be achieved by using scheduling or allocation algorithms, for large-scale applications, but such a task can be complex and error-prone. Virtual machines (VMs) represent a higher-level approach, whereby resources can be managed without dealing with lower-level details, such as scheduling algorithms, scheduling parameters and so on. However, the overhead penalty imposed by traditional VMs is unsuitable for real-time applications. On the other hand, virtualisation has not been previously exploited as a means to achieve resource adaptation. This study presents a lightweight VM framework that exploits application-level virtualisation to achieve resource adaptation in middleware for soft real-time applications. Experimental results are presented to validate the approach.

1 Introduction

The recent emergence of new application areas for middleware, such as embedded systems, real-time systems and multimedia, has imposed new challenges, which most existing middleware platforms are unable to meet. Many applications (e.g. distributed multimedia) are themselves inherently dynamic. For instance, the number of participants in a video-conference system may dynamically vary at any time. As a consequence, the resource requirements of such an application will inevitably fluctuate over time. Therefore such platforms should provide facilities to adapt (i.e. dynamically reconfigure) the resources allocated according to the perceived changes to the environment. An example of such an adaptation is a redistribution of both CPU-time and memory to the set of activities that a system performs.

Resource adaptation can be achieved by using scheduling or allocation algorithms, for large-scale applications, but such a task can be complex and error-prone. Virtual machines (VMs) represent a higher-level approach, whereby resources can be managed without dealing with lower-level details, such as scheduling algorithms, scheduling parameters and so on. Also, VMs are an effective mechanism to isolate resources. Hence, VM ensures that the allocated resources will be available when needed even in overloaded conditions. VMs have successfully been used in grid and parallel programming environments. In addition, recent trends in utility computing, including data centres, report using virtualisation as a means to optimise resource usage. However, traditional virtualisation faces two core challenges: it does not offer quality of service (QoS) guarantees and has a performance penalty that is unsuitable for certain kinds of applications [1]. In the case of highly dynamic real-time environments, we advocate using more efficient lightweight VMs as such environments may require frequent creation and deletion of VMs. Crucially, virtualisation has not been previously exploited as a means to achieve resource adaptation. We introduce an application-level virtualisation framework, called virtual task machine (VTM). Our focus is on developing an open-ended framework for lightweight virtualisation, rather than being tied to any particular VM implementation. Although a particular implementation of the framework is introduced, this implementation is only presented as a means to evaluate the framework. The main contribution of this paper is a lightweight VM-based approach to resource adaptation in middleware that achieves a better performance than traditional VMs.

The paper is structured as follows. Section 2 presents an analysis of related works. The lightweight virtual machine framework is presented in Section 3. The evaluation of the framework is shown in Section 4. Finally, some concluding remarks are presented in Section 5.

2 Related works

There has been a significant interest in the development of resource models for distributed object systems. The Object

© The Institution of Engineering and Technology 2011

Management Group's (OMG) dynamic scheduling adopted standard [2] has been released to overcome the limitations imposed by real-time CORBA (RT-CORBA) [3]. The QuO project [4] provides a framework for the specification of QoS of CORBA object interactions. A joint effort between the QuO project and the TAO project [5] is being carried out, which aims to study adaptive middleware for real-time systems [6]. The real-time adaptive resource management [7] provides middleware mechanisms for QoS negotiation and adaptation. The ERDoS project [8] offers a generic and comprehensive resource framework. dynamicTAO [9] is part of the 2K project [10], which aims at developing a distributed operating system with an integrated architecture for adaptation.

All the middleware approaches mentioned above provide some support for resource management, however, none of these approaches makes use of VMs to carry out resource allocation or resource adaptation. As said earlier, VMs have successfully been used in grid and parallel programming environments [6, 11–17]. In addition, recent trends in utility computing (including data centres) and cloud computing [18] report using virtualisation as a means to optimise resource usage. Examples of such VMs are Xen [19] and VMware [20]. However, these approaches are heavyweight VMs which provide a virtualisation of a complete operating system environment over a host operating system whose performance penalty goes from 10 to 20% [21, 22]. Even worse, this penalty is higher when there is a huge amount of input–output (IO) operations. Moreover, heavyweight VMs do not offer response time guarantees [1]. Operating system-level VMs [23, 24] are an alternative to obtain a performance near to the native operating system. However, the main drawback of this approach is that the kernel needs to be recompiled. Therefore we advocate using application-level virtualisation instead. Finally, we are not aware of any work exploiting virtualisation as a means to achieve resource adaptation. A lightweight VM framework is presented in the following section.

3 VM framework

3.1 Resource virtualisation

We focus on application-level virtualisation. This kind of virtualisation is built on top of the operating system services. The framework offers partial virtualisation. That is, instead of providing a virtualisation of the complete set of system resources, our approach virtualises only the resources that are required by a particular kind of application. The most important elements of the virtualisation framework are abstract resources, resource factories and resource managers, as depicted in Fig. 1.

Abstract resources explicitly represent system resources. There may be various levels of abstraction in which higher-level resources are constructed on top of lower-level resources. At the lowest-level are represented physical resources such as CPU, memory and network resources. This level may contain information about the type and speed of the CPU as well as the type and speed of the physical network. Higher abstraction levels then include the representation of more abstract resources such as virtual memory, team of threads, network connections and more generic type of resources. In addition, abstract resources support operations that allow them to have access to both the adjacent higher and lower level. The recursive use of these operations allows the user to navigate through all abstraction levels.

Resource managers are responsible for managing resources, that is, such managers either map or multiplex higher-level resources on top of lower-level resources. Furthermore, resource schedulers are a specialisation of managers and are in charge of managing processing resources such as threads or virtual processors (VPs) (or kernel threads). Therefore different granularity levels for resource management can be achieved.

Lastly, the main duty of resource factories is to create abstract resources. For this purpose, higher-level factories make use of lower-level factories to construct higher-level resources.

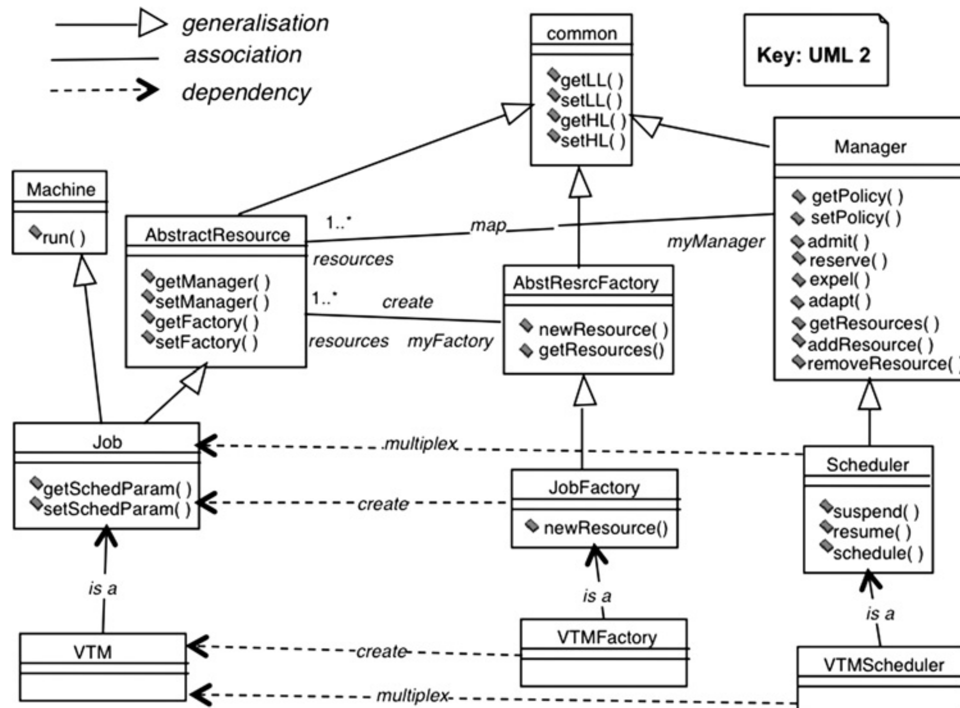

Fig. 1 *UML class diagram of the VM framework*

© The Institution of Engineering and Technology 2011
IET Softw., 2011, Vol. 5, Iss. 2, pp. 229–237
doi: 10.1049/iet-sen.2009.0091

Resource factories follow the principle of the factory method design pattern [25], whereby details involved in the instantiation of an object or component are hidden. For instance, the links to higher and lower levels of a new resource instance are set within the factory functionality.

In addition, machines are capable of performing some activity, that is, they receive messages and process them. Thus, machines may be either abstract processing resources (e.g. threads) or physical processing resources (e.g. CPUs). In contrast, jobs only refer to abstract processing resources since they inherit from the abstract resource class. Both abstract resources and jobs are created by factories. In addition, abstract resources are managed by managers. However, since jobs are processing resources, they are managed by schedulers instead. Importantly, VTMs are themselves jobs, which may encompass both lower-level jobs (e.g. threads and processes) and abstract resources (e.g. memory and network resources). A VTM provides the execution environment of a particular task or service (e.g. transcoding video files to MP4). Besides, the VTMFactory and VTMScheduler are responsible for creating and scheduling VTMs, respectively.

Some of the most relevant operations supported by the framework are introduced below. The interface of a job exposes the operations `getSchedParam()` and `setSchedParam()`. The former is in charge of accessing predefined settings. The latter is responsible for performing a control admission test. If successful, resources are reserved and the scheduling parameters are set. The manager interface exposes the operation `admit()` which performs an admission control test that determines whether or not there are enough resources to satisfy a resource request. In a successful case, resources may be reserved by using the operation `reserve()`. Reservations can then be liberated by invoking the operation `expel()`. The operation `adapt()` makes use of the two former operations to carry out a reallocation of resources.

Similar to factories, through the operation `getResources()`, resource managers are able to retrieve the references of the underlying resources they control. Such references may be included or removed from a manager's registry by using the operations `addResource()` and `removeResource()`, respectively. In addition, the management policy is obtained by accessing the operation `getPolicy()`, whereas the operation `setPolicy()` allows the user to set the management policy of the manager. Moreover, the scheduler allows the user to suspend and resume an abstract processing resource by invoking the operations `suspend()` and `resume()`, respectively. Lastly, the execution order of jobs is determined by the operation `schedule()`.

For instance, a particular instantiation of a VTM and their associated scheduler and factory is shown in Fig. 2 (note, however, that the framework does not prescribe any restriction in the number of abstraction levels nor the resource types modelled). At the top-level of the resource hierarchy is placed a VTM, which encompasses both memory buffer and a team abstraction. The team abstraction in turn includes two or more user-level threads. Moreover, a user-level thread is supported by one or more VPs, that is, kernel-level threads. At the bottom of the hierarchy are physical resources. In addition, a VTM factory is at the top of the factory hierarchy and uses both a memory and a team factory. The team factory is then supported by both the thread and the VP factory. The manager hierarchy involves the team scheduler and the memory manager, which support the VTM scheduler to suspend a VTM by temporally freeing CPU and memory resources, respectively. The thread scheduler in turn allows the team scheduler to suspend its threads. The VP scheduler supports the pre-emption of VPs. Conversely, this hierarchy also provides support for resuming suspended VTMs.

3.2 VTMs: a particular implementation

As mentioned before, a VTM is a lightweight VM which partially virtualises the resource execution environment. For example, in the case of providing support for multimedia communication, it is enough to provide a virtualisation of network and CPU resources. We developed a particular instantiation of the

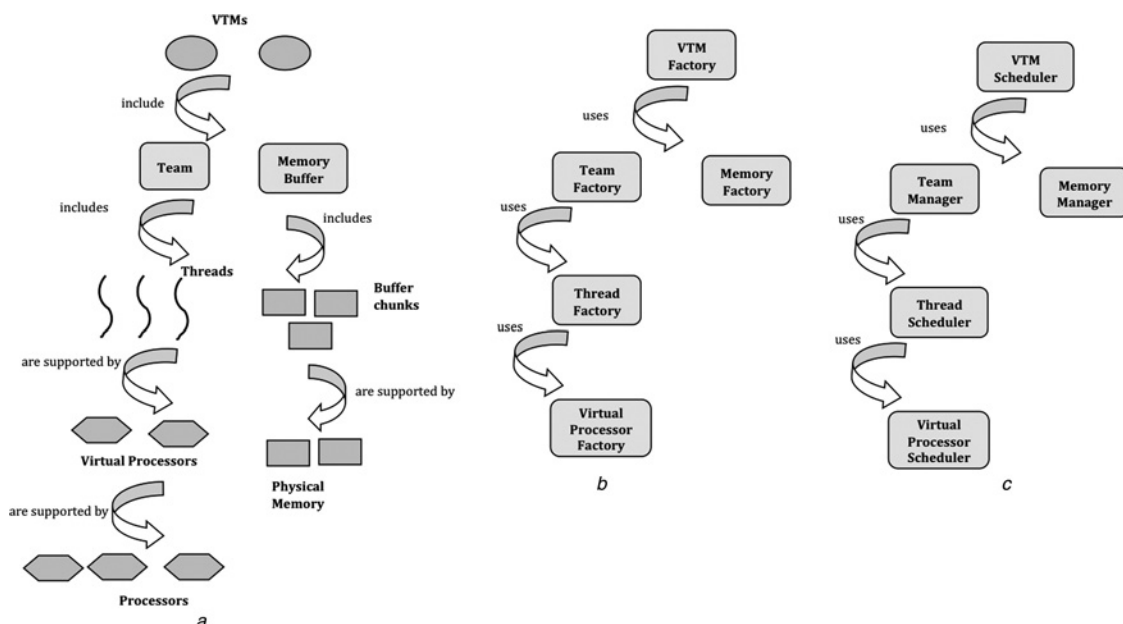

Fig. 2 *Particular instantiation of a VTM, factory and manager hierarchies*

a Hierarchy of abstract resources
b Factory hierarchy
c Manager hierarchy

© The Institution of Engineering and Technology 2011

framework which involves such kind of virtualisation. Our VM prototype is a user-space module constructed in C++ on top of POSIX threads and runs on Linux boxes. The signals SIGSTOP and SIGCONT are used to suspend and resume the execution of processes, respectively. The VTM scheduler is periodically woken up by a timer according to the quantum value. The VTM scheduler can be accessed either as a command shell or as a service from an executing program. Note, however, that the framework does not prescribe any restriction in the type of resources virtualised nor it is tied to any particular implementation.

3.2.1 Network bandwidth: Virtualisation of network bandwidth resources is achieved by controlling the bandwidth share of both incoming and outgoing bandwidth. This is implemented by using Diffserv based on a hierarchical token bucket queuing discipline [26]. This scheduling algorithm is used to limit the rate of received and sent network traffic. It is based on class services, whereby traffic can be classified and different assured rates can be associated with each class. The following service classes were defined. A 6 Mbit bandwidth is assigned to the gold service while the remaining 4 Mbits are allocated to the best effort class. At any time the spare bandwidth is allocated with the highest priority to the gold service. This configuration is achieved by marking packets at the edge router taking into account the target IP address. Such packets are forwarded to an output router, which places the packets in different queues according to the markings. The output router splits the network bandwidth into these queues. The operation `network_reserve(VTM, bandwidth, target_ip_adresses)` is used to reserve network resources. This operation involves remotely invoking a number of scripts in both routers to achieve this reservation. Such scripts use the `tc` shell command to manipulate traffic control settings. This command is part of the `iproute2` Linux package [27] already included in some distributions such as Debian and RedHat. Further description of the network configuration is presented in Section 4.1.2.

3.2.2 Computer processing unit: Virtualisation of CPU resources is achieved by employing a two-level scheduling model. The VTM scheduler is responsible for scheduling the processing resources of the VTMs. For this purpose, the VTM scheduler manages the team schedulers associated with such VTMs. The VTM scheduler uses a scheduling structure similar to [28] and extends it to a two-level scheduling structure. Instead of scheduling single jobs, a two-level structure enables scheduling job groups. This feature is required as a VTM can contain multiple jobs.

The VTM scheduler is placed at the highest level, as shown in Table 1. Team schedulers are then located at the second level of scheduling. In addition, active jobs are placed at the third highest level. Finally, non-active jobs are placed in a waiting list, a level below active jobs. The VTM scheduler wakes up periodically given a quantum time, for example, every 10 ms, to dispatch the next team scheduler by un-blocking it and then returns to sleep. The team scheduler then removes the jobs located at the third level and introduces them into the waiting list at the bottom level. Following this, the team scheduler selects one or more jobs from the waiting list and places them at the third level. Afterwards, the team scheduler blocks itself and the active jobs start executing. The first level of scheduling supports rate-monotonic [29], whereas the second scheduling level uses a round-robin policy [30]. A job may be either a thread or a process depending on the implementation of the scheduling structure.

Table 1 Priority scheduling structure

Scheduling structure	
Level	Scheduler/jobs
1	VTM scheduler
2	team scheduler
3	active jobs
4	waiting jobs

The rate-monotonic policy is realised by generating a dispatch table according to both the period and the execution time of a VTM. The dispatch table defines the execution order of VTMs. Hence, this table contains a repeatable set of time slots, whereby each slot corresponds to a slice of CPU time. Such a dispatch table is automatically generated by the VTM scheduler according to either the period and CPU usage or the period and execution time. Note that our framework is not constrained to use rate-monotonic at the first scheduling level, rather, a scheduling policy can be selected based on the environment demands. For instance, EDF scheduling policy [29] is suitable for more dynamic environments but rate-monotonic behaves better in overload conditions [31].

The operation for CPU reservation is the following:

```
cpu_reservation(VTM, period, execution_time,
  cpu_usage)
```

An example of how this operation can be employed, consider a scenario in which a video-on-demand (VOD) server transmits video streams and also carries out the transcoding of video files to MP4 [32]. Since a video transcoding job is CPU intensive, it is needed to reserve enough resources for time-sensitive operations such as stream transmission. The process of resource reservation is simple as can be seen below:

1. $vtm_{VOD_bestEffort}$.addResource(pid_transcode1)
2. $vtm_{VOD_bestEffort}$.addResource(pid_transcode2)
3. vtmSched.cpu_reservation(vtm_{VOD_gold}, 40, 16)
4. vtmSched.cpu_reservation($vtm_{VOD_bestEffort}$, 80, 8)
5. vtmSched.schedule()

The process ids of the transcoding processes are given as input parameters in order to add them into $vtm_{VOD_bestEffort}$ (lines 1 and 2). We assume that the processes in charge of the stream transmission are already bounded to vtm_{VOD_gold}. A CPU reservation for vtm_{VOD_gold} is performed in which 16 ms are assigned in a 40 ms period (line 3), whereas $vtm_{VOD_bestEffort}$ has 8 ms allocated within an 80 ms period (line 4). Hence, the CPU reservation settings shown in Table 2 are defined. As a result of performing a new schedule (line 5), the VTM scheduler

Table 2 Scheduling parameters of the VTMs

Scheduling parameters of VTMs		
VTM	Period, ms	Execution time, ms
vtm_{VOD_gold}	40	16
$vtm_{VOD_bestEffort}$	80	8

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2009.0091

generates the dispatch table shown in Table 3. Following, the VMs start executing, for example, when $vtm_{VOD_bestEffort}$ is active, the two transcoding processes share the CPU in a round-robin fashion at the second scheduling level. Again, our framework does not prescribe using round-robin; the only consideration that needs to be taken into account is that the selected scheduling policy must assure that the jobs contained in the active VTM fairly share the CPU.

Using a bare POSIX-type operating system such as Linux to achieve CPU virtualisation has two main drawbacks. First, the time-shared policy used by this type of operating system makes it impossible to know when the kernel is going to give a process the CPU even when process priorities are set, in the best case we could only assure that a process would get the CPU more often than other processes with lower priority. Secondly, although this kind of CPU reservation is achievable by using POSIX timers and signals, the complexity of using them is certainly much higher than using our API.

3.3 Resource adaptation in middleware

Although we focus on application-level virtualisation, our approach can also be used at the middleware level since the only requirement imposed by our VM is to have access to the operating system services. Our middleware offers hooks, called interceptors, which are used to co-ordinate interactions between the middleware and the VMs. Interceptors are dynamically pluggable components that may be inserted in the communication path of component and connector interconnections. Monitor interceptors are in charge of detecting QoS violations whereas resource interceptors are responsible for accessing VTMs able to allocate CPU and network bandwidth resources.

The middleware also supports control aspects which include strategy selectors and strategy activators. It is the responsibility of strategy selectors to decide which strategy to apply upon the occurrence of a QoS violation. Strategy activators are then in charge of realising the adaptation strategy by providing the detailed implementation of this strategy. Hence, the strategy selector has access to the VTM scheduler to perform resource adaptation. The VTM scheduler supports the following resource adaptation operations:

```
cpu_adapt(VTM_x, VTM_y, period_x, period_y,
  exec_time_x, exec_time_y)
```

Table 3 Dispatch table

Dispatch table	
Time slot, ms	VTM
1–8	vtm_{VOD_gold}
9–16	vtm_{VOD_gold}
17–24	idle
25–32	idle
33–40	idle
41–48	vtm_{VOD_gold}
49–56	vtm_{VOD_gold}
57–64	idle
65–72	idle
73–80	$vtm_{VOD_bestEffort}$

```
network_adapt(VTM_x, VTM_y, bandwidth_x,
  bandwidth_y)
```

4 Evaluation

4.1 Quantitative evaluation

We present a performance evaluation of the C++ prototype and compare it with Xen whose performance is comparable to VMware's [21, 22]. Also, an evaluation of the CPU adaptation capabilities of both the lightweight VM framework and Xen is performed for an experimental scenario involving a VOD middleware system.

4.1.1 VM performance: The experiments were carried out on a Pentium 4 at 1.2 GHz with 1 MB of cache memory and 3 GB of RAM memory running Linux Fedora 7 Kernel 2.6.22 with Xen 3.1.0. Xen Domain0 and the guest domains were set up with 500 MB RAM. The lightweight VM was run directly on top of the Linux operating system.

Scheduling overhead of lightweight VMs: In the case of having two VMs running, whereby each one contains two jobs, we obtain a scheduling overhead of 0.008 ms, as shown in Table 4. On the other extreme where we have 200 VMs and each VM has 200 jobs, we obtain an overhead of 0.265 ms. The overhead is acceptable for applications having execution times in the order of tens of milliseconds, such as our experimental scenarios (see below), where the performance penalty is around 1–2%.

Adaptation overhead: We also performed a number of tests to calculate the resource adaptation overhead. As shown in Table 5, the first three CPU adaptation scenarios involved running one, two and 20 VMs, respectively. In these scenarios, the lightweight VMs were given 20 ms of execution time over a 400 ms period. In the case of Xen each VM was given 20% of CPU. The fourth scenario involved 200 VMs running 20 ms of execution time over a 4000 ms period. In the latter three cases, the test for CPU adaptation consisted of giving more CPU time to a VM out of another VM. For instance, the lightweight VM_x was given 10 ms more of execution time out of VM_y. That is

Table 4 Scheduling overhead of lightweight VMs

	No VM	No jobs	Scheduling overhead, ms
1	2	2	0.008
2	2	20	0.008
3	2	200	0.008
4	20	2	0.023
5	20	20	0.050
6	20	200	0.260
7	200	2	0.039
8	200	20	0.053
9	200	200	0.265

Table 5 CPU adaptation overhead

	No VM	Lightweight VM, ms	Xen VM, ms
1	1	0.099	226.65
2	2	0.099	443.76
3	20	0.436	–
4	200	212.74	–

© The Institution of Engineering and Technology 2011

VM_x was set to 30 ms of execution time, whereas VM_y was downgraded to 10 ms. In all the cases, the VM obtaining more resources was running a video file transcoding process whereas the rest of the VMs were idle. The adaptation time was measured 30 times and the average time was given as a result. Our results show that the more lightweight VMs that are running, the higher the overhead that is imposed. Nevertheless, in the worse-case scenario where we have up to 200 VMs, we obtain an overhead of only 212.74 ms. In the second scenario, one Xen VM was downgraded from 20 to 15% of CPU, whereas the other one was upgraded from 5 to 10% of CPU. It should be noted that our lightweight VM is much faster.

In the case of the network adaptation, consisting of changing the bandwidth share of the service classes, our results show that such an adaptation takes 49.79 ms regardless of the number of lightweight VMs running.

4.1.2 VOD scenario: CPU and network adaptation: The experiments were carried out on the testbed shown in Fig. 3. The network routers Edge1 and Edge2 involve a PC Pentium III at 800 MHz, 64 MB RAM, both Edges running Linux Red Hat 7.2 Kernel 2.4.7-10. Both, the core and the video server are a Compaq 6510 laptop, Core 2 Duo T7300 at 2 GHz, 2 GB RAM. The VOD system employs MPEG-TS (MPEG-4 Part 14) and RTP. We use the VideoLAN Software [33], which is a middleware system that allows for the transmission of video streams in a distributed environment. The video transmissions are based on a 14.82 MB QuickTime (mov) file which is transcoded to MP4 (MPEG-4 multimedia container file format) [32] by the video codecs. To integrate our VM framework with the VOD system, we have added both resource and monitor interceptors to the middleware. A traffic generator is used to overload the network by injecting traffic. Resource virtualisation is carried out only in the server side in the experimental scenario.

For this experimental scenario, consider that up to five video streams can be supported by the gold class, whereas the best effort class supports the execution of the two transcoding processes and deals with the bandwidth demands of the injected traffic. The gold class provides QoS guarantees whereas the latter does not offer any guarantees. We follow the approach defined in [28] to obtain the execution time required to process a frame on the server side. In this approach a number of runs are executed on the target platform to have an application profile of the CPU demands. The number of runs is based on the central limit theorem in order to converge to a normal sampling distribution which happens to be 30. The average execution time is calculated based on $U = C/T \times 100\%$, where U is the percentage of CPU utilisation, C is the execution time and T is the period, therefore $C = (U \times T)/100$. The observed average CPU usage time to process a stream is 14.19%. In addition, the resolution of the transmitted video is 25 frames per second (fps), hence, it is required to process a frame every 40 ms. As a result, the time required to process a frame in our testbed is $C = (14.19 \times 40)/100 = 5.68$ ms. It is also obtained that the average throughput requirements of a single video stream, of the video file employed in our testbed, is 1214 kbits/s. We assume that each video file has associated a QoS profile where QoS requirements can be obtained.

We have a scenario where the VOD server is transmitting four video streams. The gold service's VM has initially allocated 24 ms of execution time in a 40 ms period of CPU resources and a network bandwidth of 5 Mbit/s. The VOD server then receives a request for an additional stream. However, the QoS manager determines the VM has not enough CPU nor network resources to process the new request. As a result, both CPU and network adaptations are carried out. In the case of CPU adaptation, 8 ms are taken out from the best effort VM so that the CPU time of the gold VM is increased to 32 ms. Given that one stream requires 5.68 ms at the server side, the selected execution

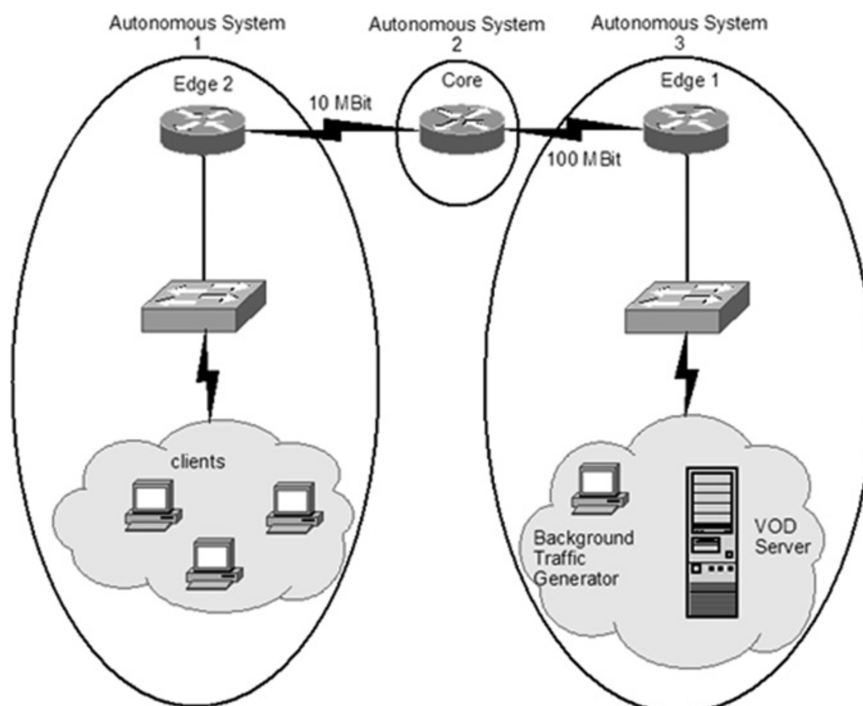

Fig. 3 *VOD system testbed*

© The Institution of Engineering and Technology 2011
doi: 10.1049/iet-sen.2009.0091

time is enough to cover the processing requirements of five streams. A single stream requires 5.68 ms of processing time; hence, five streams require 28.38 ms which can be covered by the 32 ms of execution time reserved. Regarding network bandwidth adaptation, a portion of 1 Mbit/s is moved from the best effort VM to the gold VM, thus, obtaining a network share of 6 Mbit/s. Since the average throughput requirements of a single video stream of the video file employed in our testbed is 1214 kbits/s, 6 Mbit/s are roughly enough to cover the bandwidth demands.

The resource adaptation capabilities of the framework were tested on four scenarios. In scenario 1 where neither the CPU nor network bandwidth adaptations of the VMs are enabled, both a considerable amount of packet loss and jitter are observed due to network congestion, see Fig. 4. The total transmitted packets are 6032. In scenario 2, network adaptation is performed but CPU adaptation is disabled and no loss is observed. However, jitter is still unstable. In scenario 3, the network adaptation is disabled but CPU adaptation is carried out. This causes a better jitter performance. Finally, in scenario 4, we have both CPU and network adaptations. The results are promising as the jitter and total transmitted packets are improved (8329), see Fig. 5. That is, a jitter within −20 to 20 ms range is achieved, with a packet loss of only 1.5%, and the transmitted video packets is increased above 25%.

The scenario 4 is repeated using Xen. Since Xen only supports CPU adaptation (i.e. network adaptation is not supported by Xen), the traffic injection was disabled to fairly compare the CPU adaptation capabilities. As in the

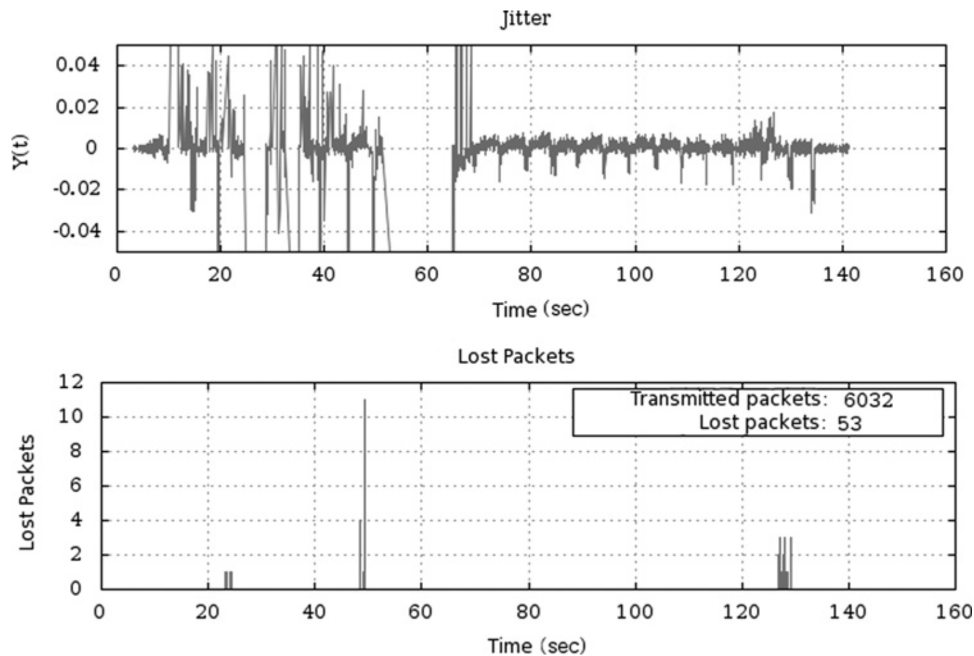

Fig. 4 *Scenario 1 – lightweight VM: jitter and packet loss*

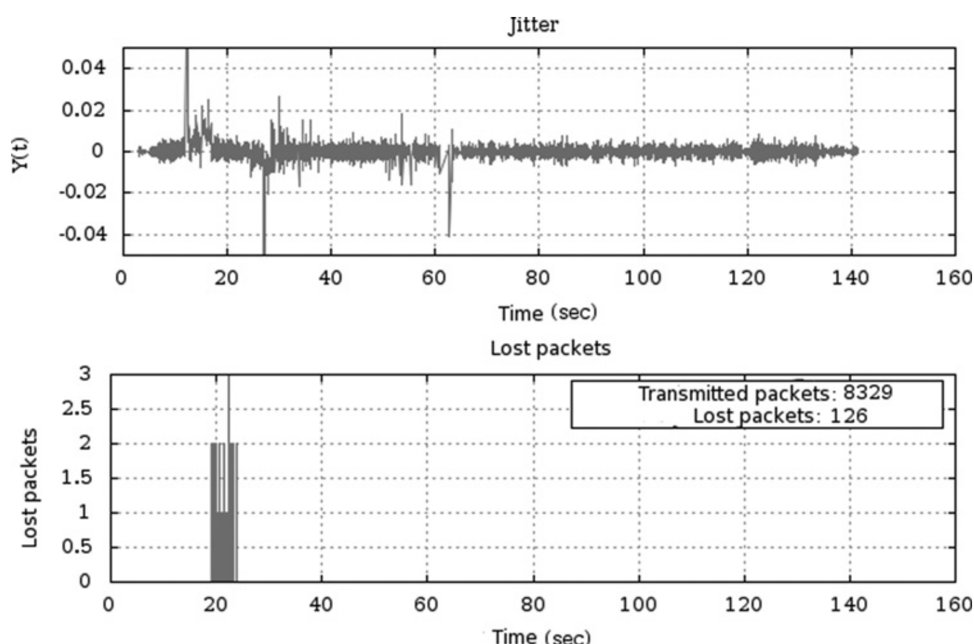

Fig. 5 *Scenario 4 – lightweight VM: jitter and packet loss*

© The Institution of Engineering and Technology 2011

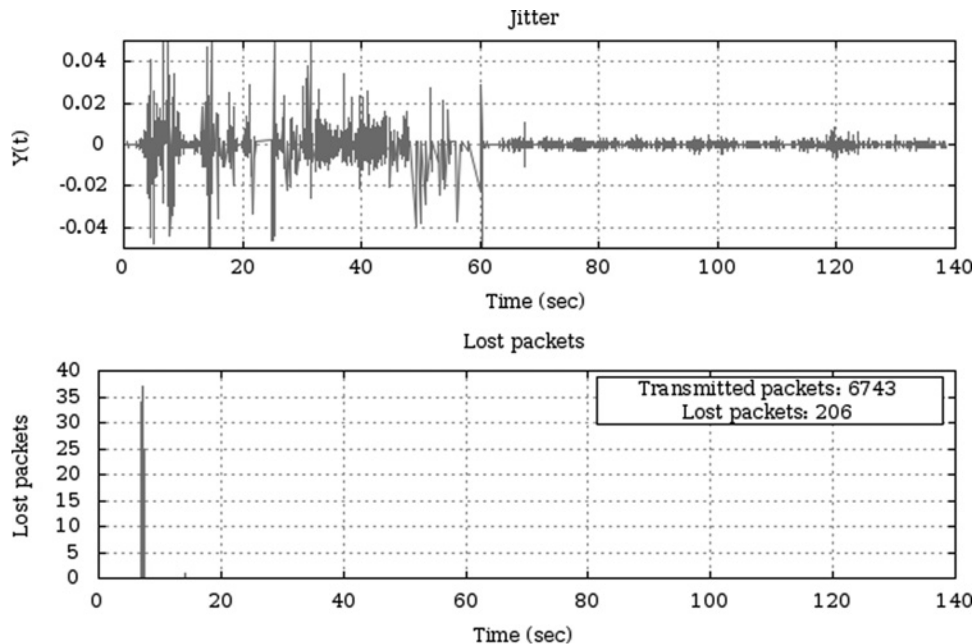

Fig. 6 *Scenario 4 – heavyweight VM Xen: jitter and packet loss*

case of the lightweight approach, the VMs are assigned to the same physical CPU, leaving the other core-free. The CPU adaptation gives 80% of CPU for the video streams and 20% for the transcoding processes (which is equivalent to the portion of CPU reserved for the lightweight VMs). A better performance is obtained by the lightweight VMs. This can be observed by comparing Figs. 5 and 6. The total transmitted packets are 20% lower in relation to the lightweight VM. The total packet loss obtained is higher. The jitter is increased in average as well as its standard deviation.

4.2 Discussion

Lightweight VMs have shown to be an effective mechanism to achieve resource adaptation in middleware. The total overhead imposed by a single lightweight VM for a CPU adaptation is about 0.099 and 49.79 ms in the case of network adaptation. In contrast, traditional VMs have a higher overhead that can go around 226.65 ms. Moreover, traditional VMs are unable to isolate the network bandwidth share.

Although, our VMs are not able to have different operating systems running on top of the host operating system, there are certain environments in which having a single operating system running is enough to cover the demands. For instance, various Linux applications can benefit from achieving resource isolation from a VM and still share the same operating system image for working properly. In this scenario, our VM can achieve a much better performance, for instance, CPU adaptation is 2000–4000 times faster.

We have shown that our lightweight VM is able to offer response time guarantees. For instance, the VOD scenario demanded a VM able to provide an execution time of 32 ms in a 40 ms period. Since our VM does not assume a hard real-time operating system underneath, eventually a job may receive less CPU than the CPU allocated due to certain kernel operations such as writing a large amount of cached data to disk. However, this kind of disruption is sporadic and tolerable by soft real-time applications as demonstrated by our experimental results.

We have shown that our approach scales well as the performance overhead of 200 VMs is only about 0.265 ms. In contrast, traditional VMs impose high overhead penalties when scaling up. For example, the main memory of the host system could be rapidly exhausted when instantiating a few VMs (e.g. 4 or 5). Also, the timer interrupts can cause a scalability issue [34]. This is because a VM must run whenever it receives a timer interrupt even if it is idle, thus, imposing a high overhead with multiple context switches.

5 Concluding remarks

We have presented an application-level and lightweight virtualisation framework. The main novelty of the framework regards the fact that it exploits virtualisation as a means to achieve resource adaptation in middleware for soft real-time applications. Our approach avoids spreading unwanted changes when a resource adaptation takes place. That is, since resources allocated to services are encapsulated in VMs, the reconfiguration of a VM does not have a negative effect on other VMs. This approach helps to prevent unpredictable behaviour on overload conditions as only the overloaded services are affected. We advocate partial resource virtualisation as there are certain kinds of applications which do not require a virtualisation of the whole resources system. For example, virtualisation of CPU and network bandwidth is enough for VOD applications. In addition, the experimental results show that our virtualisation approach is able to achieve much better performance than traditional VMs.

Ongoing work concerns about providing support for the virtualisation of multicore systems as well as other kind of resources, such as memory and disk, as well as employing lightweight virtualisation in high-performance computing.

6 References

1 Kroeker, K.L.: 'The evolution of virtualization', *Commun. ACM*, 2009, **52**, (3), pp. 18–20

2 OMG: Dynamic Scheduling, Final Adopted Specification. ptc/01-08-34. 2001, Object Management Group

3 OMG: Realtime CORBA 1.0 Adopted Specification. ptc/99-06-02. 1999, Object Management Group

© The Institution of Engineering and Technology 2011

doi: 10.1049/iet-sen.2009.0091

4 Pal, P., Loyall, J., Schantz, R., *et al.*: 'Using QDL to specify QoS aware distributed (QuO) application configuration'. Third IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'2K), Newport Beach, California, USA, 2000
5 Schmidt, D.C., Levine, D.L., Mungee, S.: 'The design of the TAO real-time object request broker', *Comput. Commun.*, 1998, **21**, (4), pp. 294–324
6 Schantz, R.E., Loyall, J.P., Rodrigues, C., *et al.*: 'Flexible and adaptive QoS control for distributed real-time and embedded middleware'. Fourth IFIP/ACM/USENIX Int. Conf. on Distributed Systems Platforms, Rio de Janeiro, Brazil, 2003
7 Huang, J., Jha, R., Heimerdinger, W., Muhammad, M., Lauzac, S., Kannikeswaran, B., Schwan, K., Zhao, W., Bettati, R.: 'RT-ARM: a real-time adaptive resource management system for distributed mission-critical applications'. Proc. IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, December 1997
8 Chatterjee, S., Sabata, B., Brown, M.: 'Adaptive QoS support for distributed, Java-based applications'. IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORK), St-Malo, France, 1999
9 Kon, F., Román, M., Liu, P., *et al.*: 'Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB'. IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing, (Middleware'2000), New York, USA, 2000
10 Kon, F.: '2K: A component-based, network-centric operating system for the next millennium' (Systems Software Research Group, University of Illinois at Urbana Champaign, 2000), http://choices.cs.uiuc.edu/2k/
11 Sundarrajan, S., Nellitheertha, H., Bhattacharya, S., Arurkar, N.: 'Nova: an approach to on-demand virtual execution environments for grids'. Sixth IEEE Int. Symp. on Cluster Computing and the Grid, CCGRID'06, 16–19 May 2006, vol. 1
12 Neogi, R.: 'Productivity and performance in parallel programming environments using a novel virtual machine framework'. 12th Int. Conf. on Parallel and Distributed Systems, ICPADS 2006, 12–15 July 2006, vol. 1
13 Petrone, M., Zarrelli, R.: 'Enabling PVM to build parallel multidomain virtual machines'. 14th Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing, PDP 2006, 15–17 February 2006
14 Selvi, S.T., Kumar, R., Balakrishnan, P., *et al.*: 'Virtual cluster development environment for grids'. 16th Int. Conf. on Advanced Computing and Communications, ADCOM 2008, 14–17 December 2008, pp. 163–169
15 Maoz, T., Barak, A., Amar, L.: 'Combining virtual machine migration with process migration for HPC on multi-clusters and grids'. IEEE Int. Conf. on Cluster Computing, 29 September–1 October 2008, pp. 89–98
16 House, B., Marshall, P., Oberg, M., *et al.*: 'Grid service hosting on virtual clusters'. Ninth IEEE/ACM Int. Conf. on Grid Computing, 29 September–1 October 2008, pp. 304–309
17 Globus: Virtual Workspaces. http://workspace.globus.org/
18 Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/
19 Barham, P., Dragovic, B., Fraser, K., *et al.*: 'Xen and the art of virtualization'. ACM Symp. on Operating Systems Principles (SOSP), 2003
20 VMware: VMware Infrastructure 3 architecture. Technical paper
21 XenSource: A Performance Comparison of Commercial Hypervisors, 2007
22 VMware: A Performance Comparison of Hypervisors, 2007
23 Banga, G., Druschel, P., Mogul, J.C.: 'Resource containers: a new facility for resource management in server systems'. Proc. 1999 USENIX/ACM Symp. on Operating System Design and Implementation, 1999
24 Linux-Vserver: http://www.linux-vserver.org
25 Gamma, E., Helm, R., Johnson, R., *et al.*: 'Design patterns, elements of object-oriented software' (Addison-Wesley, 1995)
26 Devera, M.: 'Hierarchical token bucket', Czech Republic, http://luxik.cdi.cz/~devik/qos/htb/
27 Hubert, B., Maxwell, G., van Mook, R., van Oosterhout, M., Schroeder, P.B., Spaans, J., Larroy, P.: 'Linux advanced routing and traffic control HOWTO', http://lartc.org/howto/
28 Nahrstedt, K.: 'QoS-aware resource management for distributed multimedia applications', *J. High-Speed Netw., Spec. Issue Multim. Netw.*, 1998, **7**, pp. 227–255
29 Liu, C., Layland, J.W.: 'Scheduling algorithms for multiprogramming in a hard real time environment', *J. ACM*, 1973, **20**, (1), pp. 46–61
30 Tanenbaum, A.S.: 'Modern operating systems' (Prentice-Hall, 1992)
31 Stankovic, J.: 'Implications of classical scheduling results for real-time systems', *IEEE Comput.*, 1995, **28**, (6), pp. 16–25
32 MPEG-4 part 10: 'Advanced video coding'. ITU-T
33 VideoLAN Software Project: http://www.videolan.org/
34 VMware: Timekeeping in VMware Virtual Machines. Technical Paper

© The Institution of Engineering and Technology 2011

Copyright of IET Software is the property of Institution of Engineering & Technology and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.