The Laundromat Model for Autonomic Cluster Computing

Jacob Gorm Hansen
DIKU, Univ. of Copenhagen

Eske Christiansen
DIKU, Univ. of Copenhagen

Eric Jul
DIKU/Microsoft Research

Abstract

Traditional High Performance Computing systems require extensive management and suffer from security and configuration problems. This paper presents a new cluster-management system, called Evil Man, that aims at making clusters as secure and self-managing as possible. Evil Man is inspired by real-life Laundromats: All nodes in a cluster are configured with a minimal software base consisting of a Virtual Machine Monitor and a remote bootstrapping mechanism, and customers then buy access using a simple pre-paid token scheme. All necessary application software, including the operating system, is provided by the customer as a full Virtual Machine, and boot-strapped or migrated into the cluster.

Technically, the core ingredients of Evil Man are a novel operating system Boot-strapping and Self-Migration mechanism, and a simple pre-paid Token Scheme on which all access control and accounting is based. Combined, these mechanisms create a simple, autonomous, secure, and highly flexible cluster computing platform suitable for deployment in a cluster or Grid system. We work hard at reducing the amount of privileged network-facing software on each node. For instance, the privileged part of our network protocol implementation consists of only a few hundred lines of C-code.

Performance measurements of our prototype show that Evil Man can bootstrap a non-trivial application onto an 11-node cluster in less than nine seconds and that performing live migration during application execution has negligible performance impact.

1 Introduction

The demand for access to High Performance Computing (HPC) resources is growing. Many businesses and research institutions are now running their own super-computing systems, typically clusters of commodity PC's used for tasks such as Protein Folding or Data Mining, and casual users who cannot afford their own dedicated solutions are willing to pay for access to computing power on a time-shared basis. Today most of these clusters are built and managed ad-hoc by skilled personnel, and thus are out of reach for smaller companies or users that may have only occasional needs for HPC. Grid Computing is becoming a popular way of sharing resources across institutions, but the effort required to participate in contemporary Grid systems is still fairly high, and the existence of a number of competing and seemingly-ever-evolving middleware standards has led to a fragmented community. The new challenge, as we see it, is to make corporate HPC plug-and-play, with almost instant deployment of new compute nodes and completely automated management, and at the same time allowing sharing or rental of resources without human intervention and without worries concerning the safety of potentially competing customers sharing access to the same machines.

Our system has been inspired by real-life laundromats which provide laundry facilities to customers on a simple, pay-as-you-go basis with little startup overhead. Laundromat owners need only provide the hardware and a simple token payment system while each user provides detergent, clothes, and operates the machine. Use of a token system simplifies payment and eliminates the need for a formal contractual agreement. The goal of the Evil Man[1] system is to make the sale of access to HPC resources equally convenient, and to make the installation and management of such cluster resources as simple as installing the physical hardware and supplying it with power and connectivity. As in a Laundromat, a cluster owner installs a number of machines and, using the Evil Man system, offers the machines in the cluster to customers on a pre-paid token basis.

The enabling technologies for Evil Man are a combination of Virtual Machines (VM's) and VM Migration [1, 2]. The key insight being that, from a user application viewpoint, the operating system is little more than a shared library, serving sharing and protection needs of the user application. VM's and VM migration allow Grid or HPC applications to be implemented on top of traditional operating systems with

[1] The name, Evil Man, comes from on-demand computing: if you split on-demand after the fourth letter instead of after the second then it becomes "onde mand" which, in Danish, means Evil Man.

traditional semantics. The idea is that a user will configure an application as a packaged VM instance, and then bootstrap it into, and perhaps migrate it around, a production cluster.

In a laundromat, the user pays a token for each load of clothes washed, no more, no less. There is no long term commitment to use the laundromat for the next load of clothes, nor any need to negotiate a contract for the use of the laundromat. When a token has been used, the laundromat stops providing service until another token has been provided by the user. We use the same principle in Evil Man: service is provided as long as the user provides tokens to cover the usage. At the same time, the user does not have to provide more tokens than needed to complete a given task. When an application runs out of tokens, we stop providing service.

The rest of this paper is laid out as follows: We first describe our goals and the properties of what to us comprises a useful system. From there we move gradually into more detailed descriptions of the design and implementation of the Evil Man system, including discussions on security, and on how we enforce payment for use. Finally, we include measurements of the performance of Evil Man, discuss related and future work, and conclude.

2 Goals and Principles

With our work we aim to provide autonomic and safe compute clusters by reducing to a minimum the trusted, privileged control software running on each node. We have the following goals for the Evil Man system:

Self-Management Today, the cost of hardware is often dominated by deployment and management costs. In Evil Man, it is our goal that once a new node is connected to the network and powered on, it should be ready for use and require no further management. We try to reach our goal of self-management by substantially reducing the amount of software that must be maintained on each cluster node.

Economy of Mechanism It has long been established [3] that the parts of a system that enforce protection and security should be kept as simple as possible. A program short enough that it can be understood and thoroughly reviewed by skilled programmers and security experts is less likely than a large program to contain problems that an attacker could exploit. The number of errors in software is, in general, proportional to the number of lines of code. Therefore, we work hard at reducing the amount of privileged network-facing software installed at each node.

Flexibility We have attempted to design our system in a way that puts as few restrictions on the user as possible. For example, we employ a two-stage bootstrapping process, where only the protocol for initial contact between client and compute node is specified by the system, and only the first network packet received is handled by privileged software. The bulk of the communication is performed by a user-supplied second-stage binary, inside an isolated VM compartment. The benefit here is not only stronger isolation, but also the support for arbitrary guest operating systems and network boot protocols, without needing to update any privileged or pre-installed system software on the compute nodes.

Furthermore, we attempt to adhere to the following design principles:

No Free Lunch In Evil Man, any use of a system service or resource should be accountable to and paid for by the customer using it. Otherwise, a malicious customer will be able to abuse free resources as a Denial-of-Service attack [4] against the system. One classic example of such abuse is when a TCP/IP stack allocates a small amount of system memory for every connection request, allowing an attacker to exhaust system memory by flooding the system with connection requests. With complete accountability, this type of attack can be made prohibitively expensive. Another benefit is that a system adhering to the no free lunch principle will always converge towards a steady quiescent state, because "garbage" will be automatically collected once payments run out. We happily keep garbage around as long as the user has provided payment for the service.

The End-to-End Argument Our implementation has been guided by the *end-to-end argument*[5]—that we should not provide functionality at the low level that is duplicated at higher levels. In practice this has meant that we have pushed functionality to the highest layer where it is needed.

The Principle of Least Privilege We combine the end-to-end argument with the *principle of least privilege*[3]—that each user and program should operate with the smallest possible set of privileges. For example, Evil Man nodes are TCP-accessible without having a TCP/IP stack as part of their privileged system software, and Evil Man applications migrate autonomously within an Evil Man cluster, without relying on the underlying system software providing such a facility.

3 Design and Implementation

In the following we describe the design and implementation of Evil Man.

3.1 Time-Sharing and Protection

From a customer viewpoint, the ideal is to have undivided access to a dedicated set of computing resources, as this is what provides the greatest expressive power and performance. However, many users have only occasional supercomputing needs, so dedicated computing resources often end up being idle for most of their existences. Instead, we are looking for a solution that from a customer point of view is as close as possible to having dedicated hardware, but with support for safe time-sharing across multiple users and institutions.

Compared to traditional operating systems such as UNIX, Virtual Machines have the benefit that in addition to abstracting away the particularities of the hardware and providing time-sharing, they also allow each user to specify his own operating system semantics.

Popular systems such as VMWare [6] and Xen [7] provide feature filled network control pane interfaces for remote administration. Unfortunately, complex control software may well be an Achilles Heel to the security provided by the VMM, as complexity unavoidably leads to bugs and vulnerabilities. Control pane software needs ultimate power over guest VM instances, so there is a danger that a single exploitable programming error in the control pane may jeopardize the integrity of all hosted VM's. Instead, we propose a network management model under which nodes can make their resources available to the network without resorting to the complexity of traditional VMM network-facing control panes.

Evil Man uses Xen as its underlying VMM. Xen provides time-sharing of customers' concurrently running applications, and strong isolation between them.

3.2 Preemption and Migration

When running jobs submitted by multiple users, it is hard to predict precisely the load that each job will impose on the system. Even assuming perfect advance knowledge of how each job will perform and how long it will run, choosing the optimal placement for each job subject to various service levels or deadline-guarantees, while maximizing overall revenue, is likely to be NP-hard. The situation is further complicated by unforeseeable events such as hardware failures or cases where the owner of a set of nodes wishes to run his own software at any cost. If, on the other hand, it is possible to balance the load by preempting a job during execution, *i.e.* to move it to a less loaded node, or to suspend it to disk temporarily, the problem is much easier to solve and becomes a scheduling policy issue. In this section, we describe two different means to achieving such dynamic load-balancing of a cluster, both using Live VM migration.

Live VM migration is a simple and effective technique for preemption of running jobs in a cluster. Live VM migration comes in two flavors; *Hosted Migration* and *Self-Migration*.

In hosted migration, the VMM's privileged control software contains both a TCP/IP stack and a migration service with snoop access to the memory and paging behavior of the migrating virtual machine, and the ability to destroy and create VM's at will. It is thus able to incrementally obtain a *checkpoint* of the migrating VM, and this checkpoint can be transferred over a TCP network connection, to the migration service on a another node. After the transfer is complete, the original VM may be shut down and the checkpoint resumed in a new VM at the destination. Hosted VM migration, like traditional process migration, can be optimized using a pre-copy [8] or a lazy-copy [9] technique, to make *migration downtime*[2] proportional to the size of the *writable working set* of the VM (the pages that are constantly touched, such as process stacks). The central idea in pre-copy is to migrate a checkpoint of a running process (or, in our case, a VM) by optimistically copying the process address space while the process is running, while tracking changes by means of dirty page page-fault logging. Dirty pages can then be resent to the destination node over a number of iterative transmission rounds, in the hope that each round will be shorter than the previous, so that in the end the process only has to be suspended during the final and hopefully very short retransmission round.

Because it requires the presence of a full and thus complex TCP/IP stack[3] (or a similar, reliable protocol implementation) in the privileged control software, hosted migration is not a suitable mechanism for preemption and load-balancing in our system, as this conflicts with our goal of minimizing the amount of privileged code on each node. However, the pre-copy technique may be extended in a way that allows for *self-migration* of a VM, where the VM uses its own TCP/IP stack to migrate itself somewhere else, as we shall see below.

3.2.1 Self-Migration

Intuitively, self-migration seems like an impossibility because it involves transferring a checkpoint of a running and

[2] The time from when the VM stops executing at the source node and until it resumes execution at the destination node. It is often also necessary to notify peers or switches on the local network of the move to a new hardware address, *e.g.* by sending out a broadcast ARP reply after arrival.

[3] In the Linux 2.6.12 kernel source, the `net/ipv4` directory alone contains more than 81,000 lines of code, wherein several critical bugs have been discovered over the years.

self-mutating system. However, with a simple example in mind, it is easy to convince oneself that self-migration is indeed possible: One way of performing self-migration is to reserve half of system memory for a *snapshot buffer*. A checkpoint can then be obtained simply by atomically copying all system state into the snapshot buffer. Self-migration is now simply a case of transferring the contents of this buffer to the destination node. However, this technique is unattractive for two reasons; because the checkpoint buffer wastes half of system memory, and because it will incur a downtime proportional to the size of the self-migrating system.

Instead, we use a variant of the pre-copy technique. In the following sections, we describe in detail the workings of self-migration, as we have implemented it inside Linux running on Xen, and present arguments concerning the consistency of the checkpoint that is migrated.

The writable working set of a running VM is often considerably smaller than the full VM memory footprint. This means that we can start a migration by using pre-copy over a number of rounds, and conclude it by backing up only the remaining set of dirty pages to the snapshot buffer before the final iteration. The size of this buffer can then be reduced to the size of the VM's writable working set, rather than being the size of the full VM image. To save work, we can populate the snapshot-buffer lazily, using *copy-on-write*.

The checkpointing is done iteratively. Initially, all writable virtual memory mappings held by the VM are turned into read-only mappings, and the original value of the writable bit in page table entries is recorded elsewhere, so that page faults resulting from these changes may be discerned from regular write-faults.

Tracking of changes to the checkpoint state is done upon entry to the normal Linux page fault handler. If the page fault is a write fault, and it is determined (by looking at the previously recorded original value of the writable bit) that it is a result of an otherwise legal write, the corresponding page is marked dirty in a bit vector containing entries for all pages in the system.[4] If the checkpointing has entered the final copy-on-write phase, a backup copy of the page contents is made to the snapshot buffer as well.

Our implementation consists of both a user and a kernel space component. Tasks such as TCP connection setup (or, if checkpointing to disk is the goal, opening of an appropriate disk block device), and actual transferral of checkpoint data, happen entirely in user space. The user space process interfaces the kernel space components through a special device node, `/dev/checkpoint`, and reads the checkpoint data from there.

Though the page-fault handler is the main location for tracking of changes to checkpoint state, it is not the only one. All changes to page table contents are monitored, so that during checkpointing only non-writable mappings can be created, and the page tables themselves are marked dirty when modified.

In Xen, guest operating systems see actual page tables containing physical page frame numbers, so page tables need to be remapped to point to different physical pages on the destination node. The checkpoint contains a dictionary mapping old physical frame numbers to offsets in an array of physical pages, and on arrival uses this dictionary for remapping all page tables in the VM.

A final challenge at this point is how to checkpoint state belonging to the guest VM but residing within the Virtual Machine Monitor or in DMA-accessible memory shared with hardware devices. From a checkpoint-consistency point of view, it is safe to ignore the contents of any piece of memory if that memory is not referenced from the checkpoint, or if that memory changes without causing any other changes to the state of the checkpoint. When submitting a memory page to the VMM or a device driver for external modification, we remove it from the checkpoint, and when notified by a (virtual) interrupt that the page contents have changed, we add that page to the checkpoint, marking it as dirty.

In the final round of migration, the last set of dirty pages is copied to the destination. Before a dirty page is sent, the snapshot buffer is checked to see if it contains a pristine version of that page, which in that case is sent instead. Execution can resume later from the checkpointed state, which, we claim, is consistent with the state of the VM before entering the final phase. In the next section, we substantiate this claim in detail.

3.2.2 Checkpoint Consistency

We can convince ourselves that the above technique results in a consistent checkpoint, by looking at an imaginary Virtual Machine with three pages of memory, a, b, and c. Different versions of page contents are denoted by subscripting, *e.g.*, a_0 and a_1 are subsequent values of page a, and a_0+ is an unknown value resulting from modification of a_0 while it is being copied into the checkpoint. The pages can belong to one or more of the sets S, D, B, and C. S is the set of all three pages of the Source VM, D the set of Dirty pages logged for transmission, C is the current value of the Checkpoint, and B is the set of pages backed up in the snapshot Buffer by copy-on-write.

[4] If the dirty vs. clean state of the page table entry is about to change as a result of the write, the page table page itself is marked dirty here as well.

C is a consistent checkpoint of the system S at time t, if the version number of any page in C equals the version of that page in S at time t.

Initially, all pages are dirty, and the checkpoint is empty:

$$t_0: \quad D = S = \{a_0, b_0, c_0\}, \\ C = \emptyset, B = \emptyset.$$

Pages in D are now copied to C, and, as an accidental side-effect of this copy operation, b and c happen to become dirty:

$$t_1: \quad S = \{a_0, b_1, c_1\}, D = \{b_1, c_1\}, \\ C = \{a_0, b_0+, c_0+\}, B = \emptyset.$$

We could continue copying dirty pages in D to C, in the hope that the working set would shrink further. However, we are satisfied with a working set of two pages, so we decide that the state of S at time t_1 is what we want to checkpoint.

We now activate the final phase, copy-on-write, so that dirtied pages get backed up into B before they change. After this iteration, where we again copy pages in D to C, and only c is accidentally dirtied by the copying, we have:

$$t_2: \quad S = \{a_0, b_1, c_2\}, D = \{c_2\}, \\ C = \{a_0, b_1, c_1+\}, B = \{c_1\}.$$

By transferring all pages in B to C, we end up with the desired checkpoint:

$$t_3: \quad C = \{a_0, b_1, c_1\}.$$

corresponding to the value of S at time t_1.

After the activation of copy-on-write at the start of the final phase execution of the migrating VM diverges in two directions. The original VM copy keeps running on the source node, while the migrated VM version, based on a slightly out-of-date checkpoint, continues on the destination node. Often, the original copy will be shut down shortly after migration completes, but because it is responsible for completing migration after copy-on-write is activated, some overlap of execution is necessary. If not handled properly, this leads to *external* inconsistencies. This we handle inside the VM in a very simple manner; all outgoing packets with destination IP address and port number different from those involved in the migration are dropped after the activation of copy-on-write, preserving external consistency during and after the migration.

3.3 Self-Inflation Network Protocol

We have seen that the self-migration technique allows an unprivileged VM to checkpoint itself onto another node, and this obviates the need for migration functionality in the privileged parts of our system software. However, the problem remains of receiving and resuming the checkpoint in a new VM at the destination node, without having to resort to having a receive-capable TCP server with the ability to create new Virtual Machines as part of the trusted software base there.

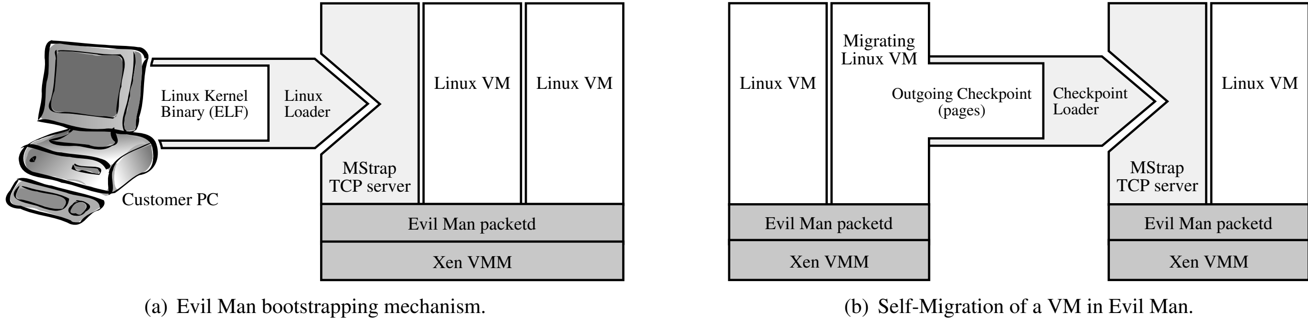
A parallel problem is the case of wishing to bootstrap a new guest VM onto an Evil Man node. In this case, we need the ability to authenticate and receive a guest VM kernel image sent via the network, and we need a way of deciding what limits to impose on its use of resources.

Our solution is a technique that we call *self-inflation*. Self-inflation involves a separate and very simplistic network protocol for requesting the creation of a new VM on a remote node. This VM contains a small and unprivileged bootloader to which a kernel image can be sent for bootstrapping, or which can be used as the target of a migration. Figure 1 shows bootstrapping and self-migration using the self-inflation protocol. In detail, the protocol has four stages:

1. The client (*e.g.* software running on the customer's PC or customer software running on another Evil Man node) first needs to find out the physical address of the node from it's IP address. In an Ethernet, this happens as in normal IP by broadcasting an Address Resolution Protocol Request (ARP Request) to the local segment. On the Client side, this *address resolution* step is normally performed by the TCP/IP stack. On the node, there is no TCP/IP stack, but Evil Man nodes understand ARP Requests, and the node configured with the requested IP address will respond with its Ethernet MAC address in an ARP Reply packet. For convenience and RFC-compliance, normal ICMP Echo Requests are also responded to with the appropriate Echo Reply.

2. When the Ethernet address is known, the client requests the creation of a new VM by sending a special IP Internet Control Message Protocol (ICMP) packet to the node. The ICMP payload carries a *boot token*, a binary structure describing the access permissions given to the new VM, and containing the initial tokens provided by the customer as payment, as described in section 3.4. The boot token is HMAC [10]-signed using a previously established shared secret, and upon successful verification of this signature, the new VM is created. Inside the new VM a small TCP server (derived from UIP [11]) is started.

3. The client connects to the TCP server inside the new VM, and uploads a binary executable (the *loader*) which fits inside the address space of the TCP server. After the loader has been successfully uploaded, and



(a) Evil Man bootstrapping mechanism.

(b) Self-Migration of a VM in Evil Man.

Figure 1. Bootstrapping and Self-Migration. Dark areas at the bottom contain privileged software.

its checksum compared to the one specified in the boot token (to prevent a man-in-the-middle from hi-jacking the new VM by tampering with the loader), the loader is given control of the TCP connection.

4. Finally, bootstrap or migration of the incoming VM can take place. The loader is responsible for the rest of the transfer, for example, for decoding an incoming kernel image and creating its virtual address space, or for receiving an incoming migration-checkpoint and restoring it to running state.

This protocol is well aligned with the no free lunch principle, though the reader might notice that we seem to be giving away ARP and ICMP replies for free. However, for both ARP and ICMP incoming and outgoing packets are of the exact same size, and so the replies are not free in the "free lunch" sense, though signed boot tokens in ICMP packets take more CPU to verify than to generate. Fortunately, our measurements show that a 2GHz Intel Pentium4, is able to verify 276,500 signatures per second, well able to keep up with a fully saturated Gigabit Ethernet link.

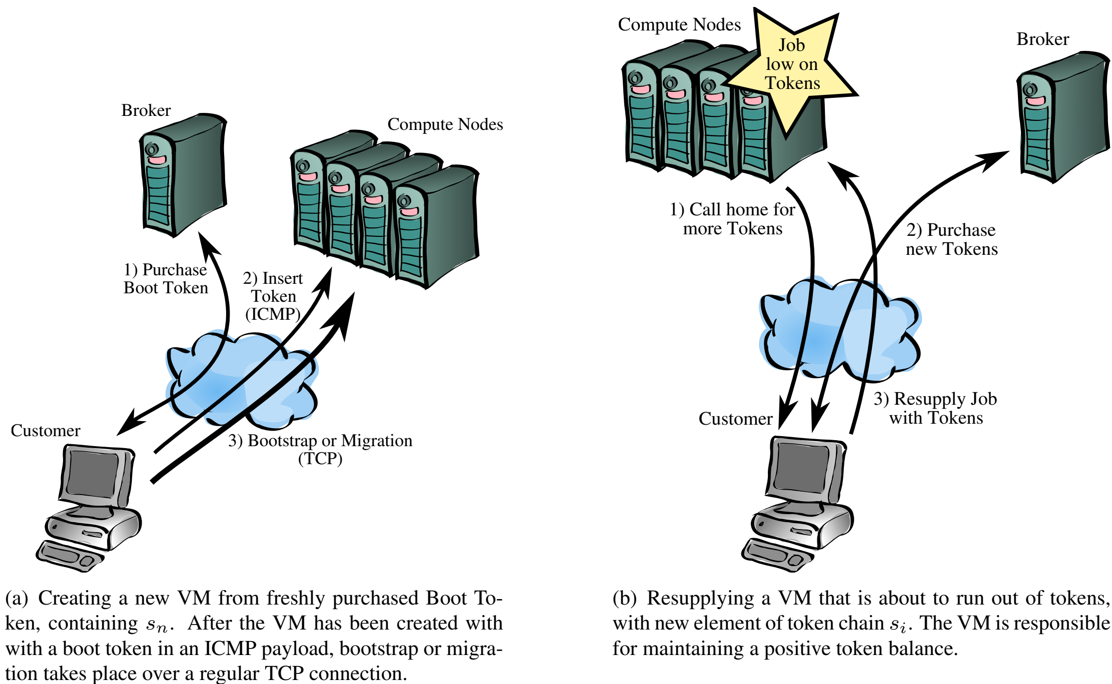
3.4 Token System

The Laundromat model is interesting when selling access to computational services, because it does not require the parties to enter into any formal agreement before work can begin. The value of each token is limited, and so the initial investment to start a new job in a VM is small. If the job makes the expected progress, more tokens can be purchased to keep it running. If the vendor fails to provide service, *e.g.*, the job stops responding or runs too slowly, the customer can move the job to a competing vendor. Individual machines need to know very little about payment: all they need to know is the number of tokens required to allow access to their resources. The actual payment plan is decoupled from the usage of the tokens; the Laundromat owner is free to use any kind of token sales or to change the policy for token sales at any time.

The Evil Man token system is designed to be as simple as possible. It builds on a so-called one-way hash chain that is hard for an attacker to reverse, but easy for the system to verify. Tokens have a fixed, low value, so that the system does not have to deal with making change, and tokens are minted for and only valid on a single node, to prevent them from being spent concurrently at two different locations.

To submit a job to a node, a customer first obtains one or more tokens from a brokering service. The customer pays the broker using a commodity payment system and gets back a boot token and a sequence of payment tokens. A token is merely a one-time password. To efficiently generate a sequence of one-time passwords we use a one-way hash function f. We start by randomly generating a single initial one-time password s_0, and derive the next password s_1 as $f(s_0)$. In general, the nth password is derived from s_0 as $f^n(s_0)$. Initially the broker establishes a sequence of, say, one million passwords ($n = 10^6$). To sell a customer the first 10 tokens, the broker tells the customer the value of s_{n-9}. These are the first tokens purchased from the chain, so the broker inserts s_n and s_{n-9} into a newly created boot token which it then signs using a secret shared with the node. The customer then sends the boot token to the node. The node verifies boot token signature, and now knows to trust s_n as the end of a valid token hash chain. It also checks that $f^{10}(s_{n-9}) = s_n$, and if so increments the customers local token balance by 10. If the customer later wishes to purchase 10 additional tokens, the broker tells the customer s_{n-19}. The customer can now send s_{n-19} to the node, which is able to verify that $f^{10}(s_{n-19}) = s_{n-9}$. If s_0 is ever reached, the broker can generate a new chain and supply the customer with a new boot token.

Tokens are consumed at a rate proportional to the customer VM's use of resources. When a customer VM reaches a negative token balance, the privileged control software on the hosting Evil Man node will respond by purging the VM from the system. To stay alive, the customer VM is responsible for keeping a positive token-balance, by purchasing



(a) Creating a new VM from freshly purchased Boot Token, containing s_n. After the VM has been created with with a boot token in an ICMP payload, bootstrap or migration takes place over a regular TCP connection.

(b) Resupplying a VM that is about to run out of tokens, with new element of token chain s_i. The VM is responsible for maintaining a positive token balance.

Figure 2. Creation of a client VM with a boot token, and subsequent resupply of tokens.

tokens from the broker, either directly or by calling on a home node to do so on its behalf. When using the latter method, the customer VM does not have to carry valuable access credentials around, and the home node will be able to monitor the progress of the VM before making further investments. Before running out of tokens, the VM is given a chance to suspend itself to disk and shut down, thereby reducing its token consumption rate dramatically. During the remaining grace period, the customer is able to revive a suspended VM by purchasing more tokens from the broker and sending them to the node. Figure 2(b), shows a customer VM calling home for an additional token supply.

Upon job completion, it is possible for the job to request a refund of any remaining tokens on the node. The node returns a signed redemption certificate that can be presented to the broker for a refund.

Double-spending of tokens is prevented by having separate hash-chains for each job and each node. Figure 2(a) shows a customer purchasing an initial boot token from the broker and with that submitting a job to a compute node. The token purchase and refund protocol is intentionally left unspecified, so that site-specific policies can be implemented.

4 Prototype Implementation

Evil Man runs on top of the Xen Virtual Machine Monitor. Our current version of Evil Man is based on the original version of Xen (the 1.x series). The performance of this version has been thoroughly measured in [7] and is often close to native execution speeds. Xen 1.x takes a pragmatic approach; device drivers reside within the Virtual Machine Monitor, and this generally leads to good performance. Recently, Xen development has taken a more microkernel-like direction, resulting in the Xen 2 [12] and recently Xen 3 series. The main architectural difference is that device drivers are now running inside virtual machines, at the cost of some context switching overhead. We have most of the components of Evil Man running on Xen 3, but parts of the self-inflation mechanism have yet to be ported to the new driver interface. For this reason, the performance results reported here are based on a late version 1.3 series Xen VMM, a development release from mid-May 2004.

Xen hosts a paravirtualized version of Linux, known as XenLinux, which we have extended with self-migration support for use with Evil Man. For Xen 1.3 we are currently deriving our work from Linux 2.4.26, where as for Xen 3 we are based on Linux 2.6.12. To turn XenLinux 2.4.26 into a self-migrating OS, we had to add or change 1,450 lines of code. Self-migration is performed entirely without VMM involvement, so we did not have to make

Benchmark	MPI-Test	MPI-FFT	DistCC
Native Linux	1746.65	128.78	347.10
Evil Man, no migr.	2127.66 †	131.56 †	353.99 †
Evil Man with migr.	2129.52 †	133.90 †	358.21 †

(values marked with † after subtracting 8.7s boot time).

Table 1. Avg. benchmark completion times.

any change to Xen itself.

Apart from the self-migration functionality inside XenLinux, Evil Man consists of a privileged network server, called `packetd`, and a specialized VM operating system, called `mstrap`. The job of `packetd` is to listen on the network for incoming ARP and ICMP packets and respond to them, and, in case the ICMP carries a payload with a valid boot token, instantiate a new VM. We use a simple keyed hash function for message authentication, as this allows us to reuse the hashing code that is also used for the token system. The main loop of `packetd` is currently 145 lines of C-code, excluding the secure hash function (currently SHA-1) which is 200 lines of code. Apart from the network driver, this is the *only* privileged network-facing code in Evil Man, and while we have not yet undertaken a full security-audit of this code, we expect this to be a realistic task.

5 Evaluation

We are currently evaluating the usefulness of our system on a small cluster of commodity PC's, and have made basic measurements of the performance of our system which we present in the following:

We have constructed three benchmarks that mimic what we expect to be typical uses of the system. One test is a parallel build of a large C-code base (the Linux 2.6 kernel) using `distcc` distributed compilation tool, and the other two are examples of scientific computations (a parallel Fast Fourier Transform (FFT), and an MPI test suite) implemented on top of LAM-MPI [13].

We run our tests on a cluster of Dell Optiplex 2GHz Intel Pentium4 (without Hyperthreading) machines, connected via Intel E1000 Gigabit Ethernet adapters. Eleven nodes from the cluster are allocated for the tests, with eight of the machines being active at the same time, and two kept as spares to use as migration-targets. The last node is acting as Master, submitting jobs to the other Slave nodes. All nodes are running the Evil Man `packetd` server inside Xen's $domain_0$ VM. Before each test, Evil Man VM's are bootstrapped with a XenLinux 2.4.26 guest kernel. In the Native Linux case, the machines are running a native 2.4.26 kernel with configuration and optimization settings similar to the guest kernels running under Xen.

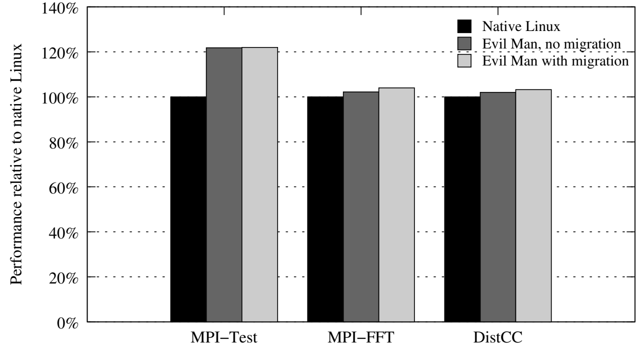


Figure 3. Relative performance.

At the start of each job, the job control script running at the Master node bootstrap a new VM onto each of the slave. The Master also acts as an NFS server, from which the slave VM's get their root file systems. In all cases each job instance is allowed the use of 200MB's of memory. Each of the tests is run in three scenarios; under native Linux directly on the hardware, in a VM under Evil Man, and in a VM under Evil Man while performing a number (4 for DistCC, 5 in the MPI tests) of randomly selected live migrations during each run. We repeat each test five times, and report average completion times after node bootstrap. Results are listed in table 1. Node bootstrap takes 8.7 seconds on average, and because this is a constant factor which will be amortized for longer running jobs, has been elided from the results.

The graph labeled DistCC in Figure 3 shows Evil Man performance relative to native Linux. Evil Man nodes take 2.0% longer to complete without migration, and 3.2% longer when four random migrations take place during the test. This test is mostly CPU-bound, suffering only a slight slowdown from the use of Evil Man on Xen. The MPI-FFT graph shows relative completion times for the MPI FFT test. In this test, native Linux also comes in ahead, with Evil Man jobs taking 2.2% (resp. 4.0% with five migrations) longer to complete on average, which we attribute to the I/O processing overhead of the Xen VMM. The final test, the LAM-MPI test suite, is mostly an I/O bound test, measuring network latencies and communication performance of an MPI setup. Compared to the other tests, Evil Man fares less well here, with a 21.8% (21.9% with 5 migrations) overhead compared to native Linux. We expect that most of this overhead is due to the way Xen tries to reduce switches between VMM and guest VM's by batching multiple network packets, at the cost of adding latency.

From the figures we can conclude that with the possible ex-

ception of very I/O-bound scenarios, the overhead of using Evil Man is not prohibitive. The extra startup time for the nodes is offset by the added flexibility and security, and, in practice, will be completely amortized for longer running jobs. The option of performing live migration of jobs, in response to load imbalances, to deal with failing hardware, or to preempt a job to let another one run, also comes at a very low cost.

6 Related Work

There is a large body on previous work on process migration, summarized in [14]. Recently, NomadBIOS [15], VMWare [1], and Xen [2], have all implemented live VM migration, of the Hosted variant, and μ-Denali [16] has implemented non-live “stop-and-copy” VM migration. Being Virtual Machine Monitors, all provide stronger isolation than traditional process migration systems, but from a security perspective, all suffer from the problem of having a complex, network-facing control interface.

The Evil Man token system is similar to the PayWord [17] micro-payment system devised by Rivest and Shamir. However, it is simplified even further by the use of shared-secret HMAC for initial token-chain signature verification, rather than full-blown public key signatures. In PayWord, the responsibility of token chain creation lies with the customer, whereas in Evil Man token chains are generated by the vendor and only released in response to continued payment for use. This removes problems of token chain revocation and return change, but increases the load on the token vendor.

Evil Man boot tokens also have similarities to to the *rcap* remote capabilities used in the PlanetLab OS [18], in that both specify limits on resource usage. Evil Man tokens are short-lived, requiring a constant supply of new tokens for the resource reservation to stay active. We believe this will lead to more fine-grained time division of resources, because there is less commitment to stay on a particular host. Because Evil Man uses Xen Virtual Machines rather than the lighter weight Linux *VServers* used in Planetlab OS, Evil Man is more flexible and more secure against kernel level security exploits, at the cost of using more resources.

7 Future Work

While Evil Man is able to host real applications, there are still a number of open issues that we need to address. Firstly, we still only have a full implementation for Xen 1.3. We are currently finalizing a port of Evil Man to the recently released Xen 3 and Linux 2.6.

Evil Man allows a user VM to access the parts of the disk that are explicitly specified in its boot token. If the VM needs to stage data to the disk, this may be handled by booting the VM from a ramdisk image and having it initially download a raw disk image from the network. Scratch space can likewise be formatted upon arrival. Our self-migration implementation does not address the migration of on-disk storage, but we have been experimenting with various ways of achieving this, including the use of RAID-1, where the VM creates a mirror of its disk on the remote node before migration, uses software RAID-1 to keep the mirror synchronized during migration, and fails over to the mirror disk after arrival.

The Evil Man token system is a mechanism for authenticating and authorizing access to resources in a decentralized fashion, but does not handle distributed resource scheduling. On top of the token system, we are working on implementing a distributed resource scheduler known as GEM [19]. GEM runs in Evil Man guest VM’s and is responsible for the minting of tokens for Evil Man nodes in the local cluster. Ultimately, GEM will also allow peering of resources across separate Evil Man clusters.

8 Conclusion

The goal of this work has been to provide paying but otherwise untrusted customers with access to cluster computing resources, while keeping policy and privileged management software to a minimum. We have leveraged the power of self-migration and self-inflation, to develop the Evil Man Laundromat system that provides a compute environment for applications that seek access to computing resources on off-the-shelf clusters of machines. With this approach, we have been able to reduce the privileged parts of the trusted software base facing the network to a few hundred lines of C-code.

We have developed a token system that provides the ability to obtain and the use of resources on a cluster in a simple, anonymous way analogous to token payment systems at Laundromats. Our token system decouples the actual payment policy from the usage of tokens; we leave it to cluster owners to devise their own policy and to implement their own sales of tokens—our system merely assures secure and reliable use of the tokens.

We present performance evaluation of using Evil Man to run non-trivial applications. Results show that Evil Man can deploy these onto a cluster with an initial startup time of less than nine seconds and that performing live migration of VM instances running parts of the application during computation adds negligible overhead. Thus performance is no

barrier to using Evil Man.

Our contributions are the simple model of the Laundromat, the use of VM self-migration and self-inflation to enable a minimal trusted software base, a token based system that provides simple and secure payment, and a working implementation.

9 Acknowledgments

We are grateful to Jørgen Sværke Hansen for his comments on previous versions of this paper. The name Evil Man was indirectly suggested to us by Fritz Henglein, who, during a talk given by Stuart Feldman, noted the relation between "Evil Man" and "On Demand". This work was supported by the Danish Center for Grid Computing, and in part by Microsoft Research.

References

[1] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005.

[2] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Networked Systems Design and Implementation NSDI '05*, May 2005.

[3] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Communications of the ACM*, July 1974.

[4] Jelena Mirkovic and Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, 2004.

[5] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[6] VMWare, Inc. *VMWare VirtualCenter Version 1.2 User's Manual*. 2004.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.

[8] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the tenth ACM Symposium on Operating System Principles*, pages 2–12. ACM Press, 1985.

[9] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 13–24. ACM Press, 1987.

[10] M. Bellare, R. Canetti, and H. Krawczyk. In N. Koblitz, editor, *Advances in Cryptology - Crypto 96 Proceedings*, volume 1109. Springer-Verlag, 1996.

[11] Adam Dunkels. Full tcp/ip for 8-bit architectures. In *Proceedings of the first international conference on mobile applications, systems and services (MOBISYS 2003)*, May 2003.

[12] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS)*, October 2004.

[13] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[14] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[15] Jacob G. Hansen and Asger K. Henriksen. Nomadic operating systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark, December 2002.

[16] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[17] Ronald L. Rivest and Adi Shamir. Payword and MicroMint: Two simple micropayment schemes. In *Security Protocols Workshop*, pages 69–87, 1996.

[18] Andy Bavier, Larry Peterson, Mike Wawrzoniak, Scott Karlin, Tammo Spalink, Timothy Roscoe, David Culler, Brent Chun, and Mic Bowman. Operating system support for planetary-scale network services. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[19] Eske Christiansen. GEM: A distributed grid broker service. Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark, January 2006.