17th ACM Symposium on Operating System Principles (SOSP'99)
Published as *Operating Systems Review* **34**(5):202–216, Dec. 1999

Design and implementation of a distributed virtual machine for networked computers

Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, Brian N. Bershad

University of Washington
Department of Computer Science and Engineering

{egs, rgrimm, artjg, bershad}@cs.washington.edu

Abstract

This paper describes the motivation, architecture and performance of a distributed virtual machine (DVM) for networked computers. DVMs rely on a distributed service architecture to meet the manageability, security and uniformity requirements of large, heterogeneous clusters of networked computers. In a DVM, system services, such as verification, security enforcement, compilation and optimization, are factored out of clients and located on powerful network servers. This partitioning of system functionality reduces resource requirements on network clients, improves site security through physical isolation and increases the manageability of a large and heterogeneous network without sacrificing performance. Our DVM implements the Java virtual machine, runs on x86 and DEC Alpha processors and supports existing Java-enabled clients.

1. Introduction

Virtual machines (VMs) have the potential to play an important role in tomorrow's networked computing environments. Current trends indicate that future networks will likely be characterized by mobile code [Thorn 97], large numbers of networked hosts per domain [ISC 99] and large numbers of devices per user that span different hardware architectures and operating systems [Hennessy 99, Weiser 93]. A new class of virtual machines, exemplified by systems such as Java and Inferno [Lindholm & Yellin 96, Dorward et al. 97], has recently emerged to meet the needs of such an environment. These modern virtual machines are compelling because they provide a platform-independent binary format, a strong type-safety guarantee that facilitates the safe execution of untrusted code and an extensive set of programming interfaces that subsume those of a general-purpose operating system. The ability to dynamically load and safely execute untrusted code has already made the Java virtual machine a ubiquitous component in extensible systems ranging from web browsers and servers to database engines and office applications. The platform independence of modern virtual machines makes it feasible to run the same applications on a wide range of computing devices, including embedded systems, handheld organizers, conventional desktop platforms and high-end enterprise servers. In addition, a single execution platform offers the potential for unified management services, thereby enabling a small staff of system administrators to effectively administer thousands or even hundreds of thousands of devices.

While modern virtual machines offer a promising future, the present is somewhat grim. For example, the Java virtual machine, despite its commercial success and ubiquity, exhibits major shortcomings. First, even though the Java virtual machine was explicitly designed for handheld devices and embedded systems, it has not been widely adopted in this domain due to its excessive processing and memory requirements [Webb 99]. Second, it is the exception, rather than the rule, to find a secure and reliable Java virtual machine [Dean et al. 97]. And third, rather than simplifying system administration, modern virtual machines, like Java, have created a substantial management problem [McGraw & Felten 96], leading many organizations to simply ban virtual machines altogether [CERT 96].

We assert that these symptoms are the result of a much larger problem that is inherent in the design of modern virtual machines. Specifically, state of the art modern virtual machines rely on the monolithic architecture of their ancestors [Goldberg 73, Popek & Goldberg 74, IBMVM 86, UCI 96]. All service components in a monolithic VM, such as verification, security management, compilation and optimization, reside locally on the host intended to run the

SOSP-17 12/1999 Kiawah Island, SC

VM applications. Such a monolithic service architecture exhibits four shortcomings:

1. Manageability: Since each modern virtual machine is a completely independent entity, there is no central point of control in an organization. Transparent and comprehensive methods for distributing security upgrades, capturing audit trails and pruning a network of rogue applications are difficult to implement.
2. Performance: Modern virtual machine services, such as authentication, just-in-time compilation and verification, have substantial processing and memory requirements. Consequently, monolithic systems are not suitable for hosts, such as embedded devices, which lack the resources to support a complete virtual machine.
3. Security: The trusted computing base (TCB) of modern VMs is not small, well-defined, or physically isolated from application code. A large TCB with ill-defined boundaries makes it difficult to construct and certify secure systems [Saltzer & Shroeder 75]. The lack of separation between virtual machine components means that a flaw in any component of the virtual machine can place the entire machine at risk [McGraw & Felten 99]. Further, co-location of VM services has resulted in non-modular systems that can exhibit complex inter-component interactions, as observed for monolithic operating systems [Accetta et al. 86, Bershad et al. 95, Engler et al. 95].
4. Scalability: Monolithic virtual machines are difficult to port across the diverse architectures and platforms found in a typical network [Seltzer 98]. In addition, they have had problems scaling over the different usage requirements encountered in organizations [Rayside et al. 98].

The goal of our research is to develop a virtual machine system that addresses the manageability, performance, security and scalability requirements of networked computing. In addition, such a system should preserve compatibility with the wide base of existing monolithic virtual machines in order to facilitate deployment. To this end, we focus on implementation techniques that preserve the external interfaces [Lindholm & Yellin 96] and platform APIs [Gosling & Yellin 96] of existing virtual machines.

We address the problems of monolithic virtual machines with a novel distributed virtual machine architecture based on service factoring and distribution. A distributed service architecture factors virtual machine services into logical components, moves these services out of clients and distributes them throughout the network. We have designed and implemented a distributed virtual machine for Java based on this architecture. Our DVM includes a Java runtime, a verifier, an optimizer, a performance monitoring service and a security manager. It differs from existing systems in that these services are factored into well-defined components and centralized where necessary.

The rest of the paper is structured as follows. The next section describes our architecture and provides an overview of our system. Section 3 describes the implementation of conventional virtual machine services under our architecture. Section 4 presents an evaluation of the architecture and Section 5 shows how a new optimization service can be accommodated under this architecture. Section 6 discusses related work; Section 7 concludes.

2. Architecture overview

The principal insight behind our work is that centralized services simplify service management by reducing the number and geographic distribution of the interfaces that must be accessed in order to manage the services. As illustrated by the widespread deployment of firewalls in the last decade [Mogul 89, Cheswick & Bellowin 94], it is far easier to manage a single, well-placed host in the network than to manage every client. Analogously, we break monolithic virtual machines up into their logical service components and factor these components out of clients into network servers.

The service architecture for a virtual machine determines *where*, *when* and *how* services are performed. The location (i.e. where), the invocation time (i.e. when), and the implementation (i.e. how) of services are constrained by the manageability, integrity and performance requirements of the overall system, and intrinsically involve engineering tradeoffs. Monolithic virtual machines represent a particular design point where all services are located on the clients and most service functionality, including on the fly compilation and security checking, is performed during the run-time of applications. While this paper shows the advantages of locating services within the network, changing the location of services without regard for their implementation can significantly decrease performance as well. For instance, a simple approach to service distribution, where services are decomposed along existing interfaces and moved, intact, to remote hosts, is likely to be prohibitively expensive due to the cost of remote communication over potentially slow links and the frequency of inter-component interactions in monolithic virtual machines. We describe an alternative design where service functionality is factored out of clients by partitioning services into static and dynamic components and present an implementation strategy that achieves performance comparable to monolithic virtual machines.

In our distributed virtual machine, services reside on centralized servers and perform most of their functionality statically, before the application is executed. Static service

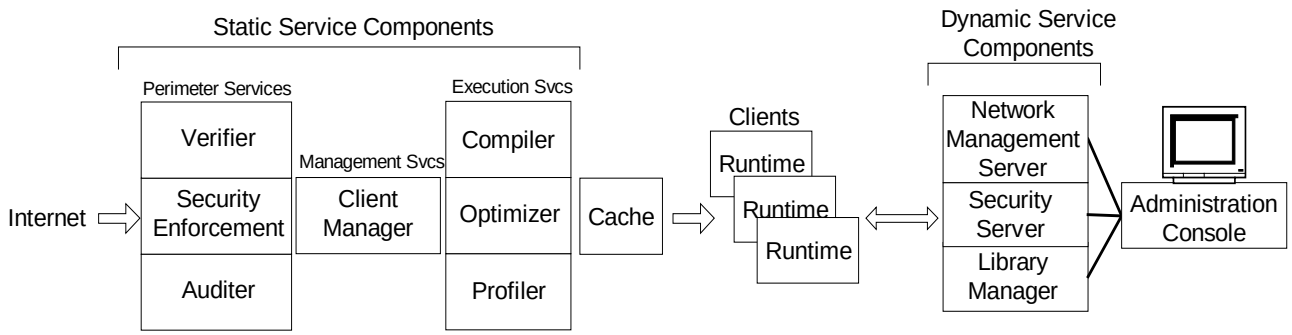

Figure 1. The organization of static and dynamic service components in a distributed virtual machine.

components, such as a verifier, compiler, auditor, profiler, and optimizer, examine the instruction segment of applications prior to execution to ensure that the application exhibits the desired service properties. For example, a verifier may check the code for type-safety, a security service may examine the statically determinable arguments to system calls, and an optimizer may check code structure for good performance along a particular path.

The dynamic service components provide service functionality during the execution of applications. They complement static service components by providing the services that inherently need to be executed at application run-time in the context of a specific client. For example, a security service may check user-supplied arguments to system calls, a profiler may collect run time statistics, and an auditing service may generate audit events based on the execution of the application.

The glue that ties the static and dynamic service components together is binary rewriting. When static service components encounter data-dependent operations that cannot be performed statically, they insert calls to the corresponding dynamic service components. For example, our static verification service checks applications for conformance against the Java VM specification. Where static checking cannot completely ascertain the safety of the program, the static verifier modifies the application so that it performs the requisite checks during its execution. The resulting application is consequently self-verifying because the checks embedded by the static service component are an integral part of the application code.

Figure 1 illustrates our distributed virtual machine architecture. Static service components produce self-servicing applications, which require minimal functionality on the clients. Dynamic service components provide service functionality to clients during run-time as necessary. The static services in our architecture are arranged in a virtual pipeline that operates on application code, as shown in Figure 2.

A distributed service architecture allows the bulk of VM service functionality to be placed where it is most convenient. A natural service placement strategy is to structure the static service components as a transparent network proxy, running on a physically secure host. Placed at a network trust boundary, like a firewall, such a proxy can transparently perform code transformations on *all* code that is introduced into an organization. In some environments, the integrity of the transformed applications cannot be guaranteed between the server and the clients, or users may introduce code into the network that has not been processed by the static services. In such environments, digital signatures attached by the static service components can ensure that the checks are inseparable from applications [Rivest et al. 78, Rivest 92], and clients can be instructed to redirect incorrectly signed or unsigned code to the

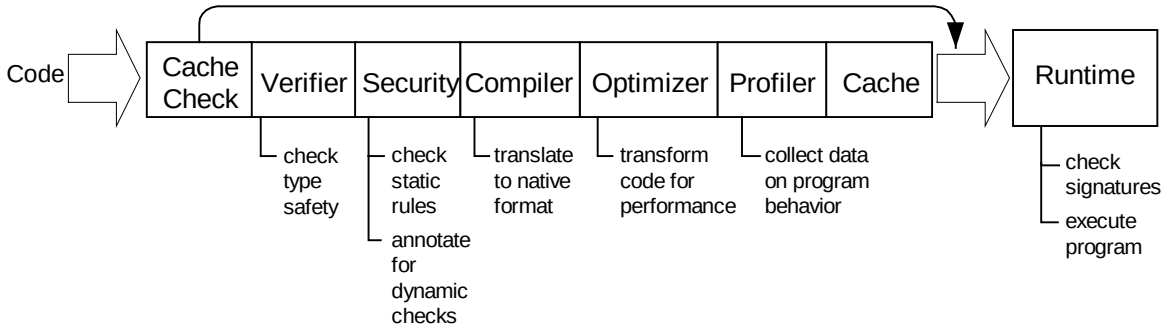

Figure 2. The flow of code through a pipeline of static service components in a distributed virtual machine. The ordering of services in this pipeline may be modified to suit organizational or functional requirements. Further, the client runtime may communicate with the static service components for client-specific services.

centralized services [Spyglass 98].

A DVM introduces a modest amount of new functionality into the existing trusted computing base of an organization. A DVM client needs to trust that the static and dynamic service components it relies on for safety, including the proxy and binary rewriter, are implemented correctly. In addition, any service authentication scheme used in the clients, which may include a digital signature checker and a key manager, form part of the trusted computing base under our design. However, we believe the actual impact of these additions to the TCB to be small. Monolithic clients already trust all of the service components that form a traditional VM and often already have provisions for cryptographic protocols and digital signature checking to support their target applications [Gong 99]. Overall, a modest increase in the TCB enables DVM clients to migrate the trusted components to physically secure, professionally managed and administered hosts, which is critical to addressing the operational problems that have plagued monolithic VMs.

Our service architecture is unique in several fundamental ways. First, the centralized services are mandatory for all clients in an organization. For example, security checks injected into incoming code are inseparable from applications at the time of their execution and are thus binding throughout the network. Second, there is a single logical point of control for all virtual machines within an organization. In the case of the security service, policies are specified and controlled from a single location; consequently, policy changes do not require the cooperation of unprivileged users. And third, using binary rewriting as a service implementation mechanism preserves compatibility with existing monolithic virtual machines. A monolithic virtual machine may subject the rewritten code to redundant checks or services, but it can take advantage of the added functionality without any modifications.

While a distributed service architecture addresses the problems faced by monolithic virtual machines, it may also pose new challenges. Centralization can lead to a bottleneck in performance or result in a single point of failure within the network. These problems can be addressed by replicated or recoverable server implementations. The next section shows how the separation between static and dynamic service components can be used to delegate state-requiring functionality to clients. Section 4 shows that this implementation strategy does not pose a bottleneck for medium sized networks even in the worst possible case and can easily be replicated to accommodate large numbers of hosts.

3. Services

We have implemented the architecture described in the previous section to support a network of Java virtual machines (JVMs). In this section, we describe the implementation of conventional virtual machine services under our architecture and show that the distributed implementation of these services addresses the shortcomings of monolithic VMs outlined in the first section. Our services are derived from the Java VM specification, which broadly defines a type-safe, object-based execution environment. Typical implementations consist of a verifier, which checks object code for type-safety, an interpreter and a set of runtime libraries. In some implementations, the interpreter is augmented with a just-in-time compiler to improve performance. The following sections describe the design and implementation of the services we have built to supplant those found in traditional Java virtual machines.

All of our services rely on a common proxy infrastructure that houses the static service components. The proxy transparently intercepts code requests from clients, parses JVM bytecodes and generates the instrumented program in the appropriate binary format. An internal filtering API allows the logically separate services described in this section to be composed on the proxy host. Parsing and code generation are performed only once for all static services, while structuring the services as independent code-transformation filters enables them to be stacked according to site-specific requirements [Heidemann & Popek 94, O'Malley & Peterson 92]. The proxy uses a cache to avoid rewriting code shared between clients and generates an audit trail for the remote administration console. The code for the dynamic service components resides on the central proxy and is distributed to clients on demand.

While the implementation details of our virtual machine services differ significantly, there are three common themes among all of them:

- Location: Factoring VM services out of clients and locating them on servers improves manageability by reducing replicated state, aids integrity by isolating services from potentially malicious code and simplifies service development and deployment.
- Service Structure: Partitioning services into static and dynamic components can enhance performance by amortizing the costly parts of a service across all hosts in the local network.
- Implementation Technique: Binary rewriting is used to implement services transparently. Binary rewriting services can be designed to incur a relatively small performance overhead while retaining backward-compatibility with existing clients.

3.1 Verification

A comprehensive set of safety constraints allows a virtual machine to integrate potentially malicious code into a privileged base system [Stata & Abadi 98, Freund &

Mitchell 98]. Indeed, Java's appeal for network computing stems principally from its strong safety guarantees, which are enforced by the Java verifier.

The task of verifying Java bytecode has been a challenge for monolithic virtual machines. First, since the Java specification is not formal in its description of the safety axioms, there are differences between verifier implementations. Verifiers from different vendors differ on underspecified issues such as constraints on the uses of uninitialized objects, subroutine calls, and cross-validation of redundant data in class files. Second, monolithic implementations tie the verifier to the rest of the VM, thereby prohibiting users from using stronger verifiers where necessary. Furthermore, monolithic verifiers make it difficult to propagate security patches to all deployed clients in a timely manner. As a case in point, 15% of all accesses to our web site originate from out-of-date browsers with well-known security holes for which many patches have been issued. Finally, the memory and processing requirements of verification render monolithic VMs unsuitable for resource limited clients, such as smart cards and embedded hosts [Cohen 97]. Some monolithic virtual machines for embedded and resource-limited systems have abandoned verification altogether for a restricted extension model based on trust [HP 99].

We address these shortcomings by decoupling verification from the rest of the VM, migrating its functionality out of clients into a network verification service and centralizing the administration of this service. Moving verification out of clients poses some challenges, however, because parts of the verification process require access to client namespaces and have traditionally required close coupling with the client JVM. Specifically, Java verification consists of four separate phases. The first three operate on a single class file in isolation, respectively making sure that the class file is internally consistent, that the code in the class file respects instruction integrity and that the code is type-safe. The fourth phase checks the interfaces that a class imports against the exported type signatures in its namespace, making sure that the assumptions that the class makes about other classes hold during linking.

In our implementation, the first three phases of verification are performed statically in a network server, while the link-time checks are performed by a small dynamic component on the client. This partitioning of functionality eliminates unnecessary communication and simplifies service implementation. During the processing of the first three phases, the verification service collects all of the assumptions that a class makes about its environment and computes the scope of these assumptions. For example, fundamental assumptions, such as inheritance relationships, affect the validity of the entire class, whereas a field reference affects only the instructions that rely on the reference. Having determined these assumptions and their scope, the verification service modifies the code to perform the corresponding checks at runtime by invoking a simple service component (Figure 3). Since most safety axioms have been checked by this time, the functionality in the dynamic component is limited to a descriptor lookup and string comparison. This lazy scheme for deferring link phase checks ensures that the classes that make up an application are not fetched from a remote, potentially slow, server unless they are required for execution.

The distributed verification service propagates any errors to the client by forwarding a replacement class that raises a verification exception during its initialization. Hence, verification errors are reflected to clients through the regular Java exception mechanisms. Since the Java VM specification intentionally leaves the time and manner of verification undefined except to say that the checks should be performed before any affected code is executed, our

```
class Hello {
    static boolean __mainChecked = false; // Inserted by the verifier
    public static void main() {
        if(__mainChecked == false) {  // Begin automatically generated code
            RTVerifier.CheckField("java.lang.System","out",
                                  "java.io.OutputStream");
            RTVerifier.CheckMethod("java.io.OutputStream","println",
                                   "(Ljava/lang/String)V");
            __mainChecked = true;
        }                              // End automatically generated code
        System.out.println("hello world");
    }
}
```

Figure 3. The hello world example after it has been processed by our distributed verification service. The vast majority of safety axioms are checked statically. Remaining checks are deferred to execution time, as shown in italics. The first check ensures that the `System` class exports a field named "`out`" of type `OutputStream`, and the second check verifies that the class `OutputStream` implements a method, "`println`," to print a string. The rewriting occurs at the bytecode level, though the example shows equivalent Java source code for clarity.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.