Improving the Security, Scalability, Manageability and Performance of System Services for Network Computing

Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, Nathan Anderson, Brian N. Bershad
{egs,rgrimm,artjg,nra,bershad}@cs.washington.edu
http://kimera.cs.washington.edu
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350

Abstract

Modern virtual machines, such as Java and Inferno, are emerging as network computing platforms. While these virtual machines provide higher-level abstractions and more sophisticated services than their predecessors from twenty years ago, their architecture has essentially remained unchanged. State of the art virtual machines are still monolithic, that is, they are comprised of closely-coupled service components, which are thus replicated over all computers in an organization. This crude replication of services forms one of the weakest points in today's networked systems, as it creates widely acknowledged and well-publicized problems of security, manageability and performance.

We have designed and implemented a new system architecture for network computing based on distributed virtual machines. In our system, virtual machine services that perform rule checking and code transformation are factored out of clients and are located in enterprise-wide network servers. The services operate by intercepting application code and modifying it on the fly to provide additional service functionality. This architecture reduces client resource demands and the size of the trusted computing base, establishes physical isolation between virtual machine services and creates a single point of administration. We demonstrate that such a distributed virtual machine architecture can provide substantially better integrity and manageability than a monolithic architecture, scales well with increasing numbers of clients, and does not entail high overhead.

1. Introduction

Virtual machines (VMs) have evolved significantly in the last two decades and may soon serve as a widely available network computing platform [Lindholm&Yellin96, Inferno, Adl-Tabatabai et al. 96]. They are particularly well suited for network computing because they offer a uniform programming model, work on a wide range of systems, and provide a variety of high-level services not supported by native architectures and general purpose operating systems. Modern virtual machines offer services, such as dynamic extensibility, verification, just-in-time compilation, configurable security policies and garbage collection, which are much more sophisticated compared to their predecessors [IBMVM86] and not readily found in general purpose operating systems [Custer 93].

In addition to the growth in the complexity of virtual machine services, the scale of deployment for VM systems has changed as well. Unlike early virtual machine systems that were typically confined to a few dedicated mainframes per enterprise, modern virtual machines are deployed in organizations with hundreds or thousands of heterogeneous hosts. Active content is now pervasive on the Internet, where about 1% of the roughly 125 million pages indexed by AltaVista reference a Java applet. More than 90% of the approximately 120 million deployed web browsers contain the Java virtual machine, and transparently fetch and execute active content from the world wide web.

Even though virtual machine services have become much more numerous and complex, and even though the scale of deployment for VM systems has changed drastically, the service architecture of virtual machines has remained unchanged over the last few decades. Today's virtual machines still rely on a monolithic architecture in which all service components reside locally on the host intended to run the applications. Consequently, service implementations and service state are replicated across all virtual machines in an organization.

As a result of this crude placement and replication of functionality, modern virtual machines suffer from security problems [Dean et al. 97], are difficult to manage, and impose high resource requirements [Madany 96]. Furthermore, colocation of VM services has resulted in non-modular systems that exhibit complex inter-component interactions, as observed [Accetta et al. 89, Bershad et al. 95, Engler et al. 95] for other monolithic systems. In particular, networks of monolithic virtual machines exhibit the following shortcomings:

- The lack of separation between virtual machine components means that a flaw in a single component of the virtual machine places the entire machine at risk. Furthermore, since policy specification and security enforcement are performed on the same host that runs potentially untrusted applications, one-time security holes can lead to long-term security compromises [Thompson 84].
- Since each virtual machine is a completely independent entity, there is no central point of control in an enterprise. There are no transparent and comprehensive techniques for distributing security upgrades, capturing audit trails, and pruning a network of rogue applications.
- Virtual machine services, such as just-in-time compilation and verification, have substantial processing and memory requirements. When performed on the client, they can reduce overall application performance.
- Monolithic systems are not suitable for hosts, such as embedded devices, which lack the resources to support a complete virtual machine.

In the rest of this paper, we describe a virtual machine architecture based on distributed service components that addresses these problems. Our architecture is founded on factoring virtual machine services into logical components and placing these services at appropriate locations in the network. We have designed and implemented a Java virtual machine based on this architecture. Our VM includes a Java runtime, a verifier, a security service, a generic binary rewriting service and a compiler. 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. Section 2 explains our approach and the goals of our architecture. Section 3 provides an overview of our system, and Section 4 describes the individual services in detail. Section 5 presents an evaluation of the architecture. Section 6 discusses related work, and Section 7 concludes.

2. Approach and Goals

Our approach to the problems posed by monolithic virtual machines is based on service decomposition and distribution. We identify the logical services in existing VMs, factor them into separate modules with well defined interfaces, and place related services at locations in the network that suit their function [Figure 1]. We identify three major categories for related services. First, the runtime provides fundamental virtual machine functionality such as interpreting bytecodes and implementing core libraries. Second, rule checking ensures that the code executed by the runtime respects a set of requisite constraints, such as typesafety and access limits. Third, code transformation changes the code to be executed, for example by translating it into a native representation, or by modifying its runtime behavior.

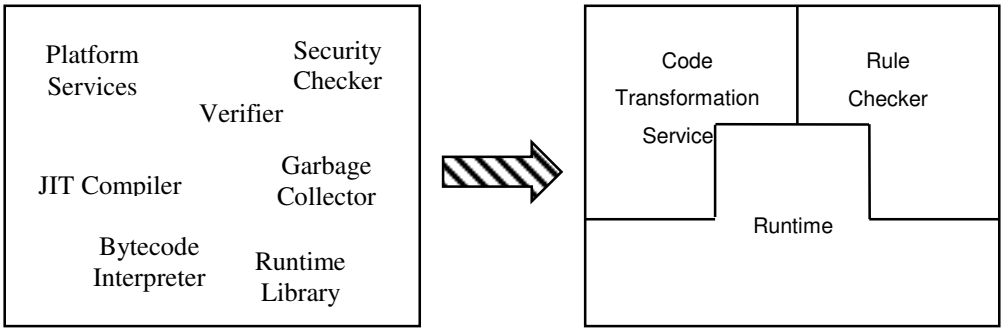


Figure 1. *Monolithic versus factored virtual machines. The diagram on the left illustrates the current state of the art for modern virtual machines. Services in a monolithic VM are often indiscernible and not isolated from each other. The diagram on the right illustrates our approach. We factor the virtual machine into three major service groups; namely, rule checking, code transformation and runtime services.*

Factoring a monolithic VM into individual services enables a modular service infrastructure, with clear component boundaries and explicit interaction. Modularization thus helps define the trusted computing base, enables the piecewise testing of components, simplifies auditing, and thereby can produce systems of high assurance. It also enables us to migrate functionality out of clients into locations more suitable to their function. This flexibility in service placement reduces the size of the trusted computing base, decreases resource requirements in clients, and physically isolates services from each other.

Service granularity and placement are determined by the security, manageability, performance and scalability requirements of the overall system. For general-purpose network computing, which forms the application domain of our architecture, these requirements can be summarized as follows:

- Security: The trusted computing base should be well defined, small, and physically isolated from application code [Saltzer&Schroeder 75]. An organization should have the ability to make network-wide, mandatory access control decisions, and know that they are being enforced on all clients.
- Manageability: Management of virtual machines should be uniform across platforms and there should be a central point of control for administration.
- Performance: Services should place a minimal processing burden on client machines and not require a large investment in the service infrastructure.
- Scalability: Virtual machine implementations should scale over the diverse architectures and platforms found in a typical network. The minimum memory and processing requirements of a virtual machine should be small, though the system should be able to utilize all available resources when necessary.

In addition to these goals, a viable architecture should be backward compatible with the large base of currently deployed monolithic virtual machines. Its implementation should preserve the external representations of existing virtual machines [Lindholm&Yellin96], as well as their platform APIs [Gosling&Yellin 96. This approach enables an upgrade path from existing monolithic virtual machines to a distributed service architecture.

3. Architecture Overview

The central tenet of our architecture is to break up monolithic virtual machines into logical services, each of which performs a single function. The result of applying this principle on a modern virtual machine is illustrated in Figure 2. The services that perform rule checking and code transformation are separated from each other as well as from the runtime. This separation between components makes it possible to place services on hosts that suit their function. In particular, we centralize *all* functionality that is not inherently part of the client runtime. The resulting service infrastructure is uniform across different platforms and provides a central point of control within an organization.

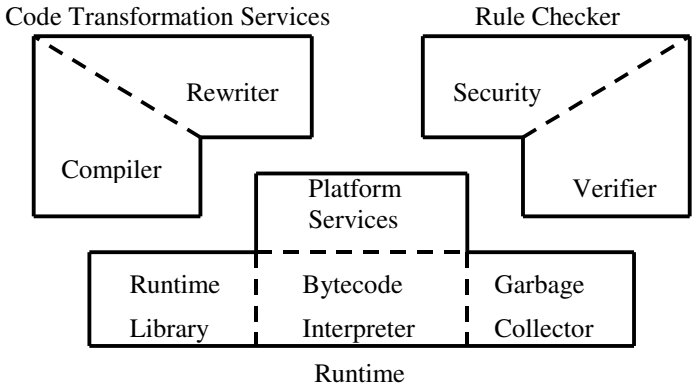


Figure 2. *The static service decomposition of the Java virtual machine.*

Centralized services in our architecture share the same basic structure. A static component performs most of the overall service functionality before an application is actually executed. Fundamentally, static components inspect incoming code and, if necessary, modify it to reflect required service characteristics. They may also inject dynamic service operations which are then executed by a small component on the client. For example, our verifier performs all code inspection, including data-flow and dependency analysis, in the static component. It defers cross-module

checks that can not be resolved to execution time. This flow of code through the service infrastructure to the client runtime is illustrated in Figure 3. The static service components generate a set of desired properties, and feed both the code and the properties to the code transformation services. After transformation, the modified code reflects an organization's policies and is thus self-managing during execution.

A generic binary rewriter and a compiler provide code transformation services. The rewriter instruments binaries with service-specific code snippets. It is structured around an event-based model. Events are associated with code abstractions, such as methods, basic blocks and individual instructions. The rewriter invokes registered event-handlers for each of the constructs it encounters in the code. An event handler may then add, change or delete instruction sequences in order to provide service functionality. The use of an event-based interface reduces system latency as it enables rewriting to be performed concurrently with the fetching and forwarding of application code.

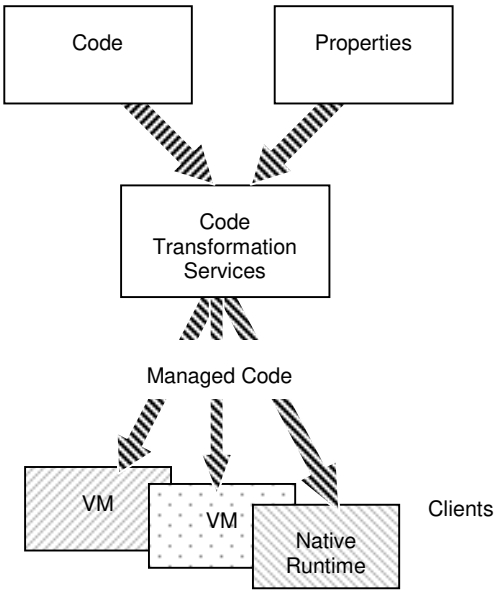


Figure 3. *Code flow in our system architecture.*

The client runtime executes the managed code produced by the code transformation services. It relies on digital signatures to verify that incoming code has been vetted by the requisite services. The runtime can redirect incorrectly signed or unsigned code to the centralized services [Spyglass 94]. Since the runtime is shielded by the centralized services, it may implement richer interfaces than those available publicly. As these interfaces may be unsafe, the verifier ensures that incoming code does not reference them. For example, our runtime provides a call to explicitly free a given object. A service for reducing garbage collection overhead may then perform liveness analysis on incoming code and insert explicit calls to free unreachable objects.

Our service architecture differs from the state of the art in two 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. They are thus binding throughout the network. Second, there is a single point of control for all virtual machines within an organization. For example, the security policy is specified and controlled at a single location. In addition, use of binary rewriting preserves compatibility with existing monolithic virtual machines. It thus provides a gradual conversion path to distributed virtual machines. A monolithic virtual machine may subject the rewritten code to redundant checks or services, but it also benefits from the added functionality.

4. Services

We have built a Java virtual machine using our distributed service architecture. Decomposing the services found in a modern Java VM, and distributing them within a network while retaining and improving their functionality posed a number of challenges. In the sections below, we discuss each of the services in detail, and report on our experience with their design and implementation using our architecture.

4.1. Verification

Java's appeal for network computing stems principally from its strong safety guarantees. A comprehensive set of safety constraints allows the VM to integrate potentially malicious code into a privileged base system [Stata&Abadi 98, Freund&Mitchell 98]. The enforcement of these type and system safety constraints is performed by the Java verifier.

A Java verifier needs to check four different classes of constraints. The first set of safety axioms ensures that the class file is well formed. For instance, indices and offset references that appear in various constructs in the class file must be within particular bounds to be valid. The second set of safety constraints makes sure that the code within a class is well structured. For example, all instructions must contain valid operands, methods must properly return control on all execution paths and control flow instructions should not branch into the middle of an instruction. The most complex part of verification involves the third set of safety axioms related to typesafety. To ensure that code is

typesafe, the verifier relies on type-inference and data flow analysis to make sure that no path of execution can result in a type error at runtime. Finally, a fourth set of safety axioms ensures that assumptions made in one class about the public interfaces of other classes are consistent with their implementation. Whereas the first three sets of constraints apply to a single class in isolation, the fourth set applies to the composition of classes.

Commercial verifiers found in monolithic virtual machines have had difficulty in imposing these constraints. First, since the Java specification is not formal in its description of the safety axioms, there are differences between verifier implementations. Verifiers from Sun and Microsoft disagree on underspecified issues such as constraints on uninitialized objects, subroutine calls, and cross-validation of redundant data in a class file. Furthermore, in response to discoveries of security flaws, monolithic VM vendors have difficulty propagating security patches to all of their clients in a timely manner. Finally, the data-flow analysis stage of verification is memory and processing intensive, and renders monolithic systems unsuitable for resource limited systems such as smart cards.

To address these shortcomings, we centralize the bulk of the verification service in accordance with our architecture. In our implementation, the first three phases of verification are performed statically in a network server. The static verifier component examines each class as it is fetched, and checks to make sure that it is safe in isolation. The fourth phase of verification requires access to dynamic client state describing the composition of classes in different namespaces. While it is possible to track and mirror this state in a network server, or to devise a protocol for querying client state remotely from the server, both techniques require modifications to existing clients. In order to remain compatible with existing virtual machines, we defer the link phase checks to a small client component. 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 the assumptions and their scope, the verification service modifies the code to perform the checks at runtime by invoking a simple service component [Figure 4]. Since most safety axioms have been resolved by this time, the functionality in the dynamic component is limited to a descriptor lookup and string comparison. The dynamic component itself resides on a central server and is provided to clients on demand. This lazy scheme for deferring link phase checks as late as possible has the property that the classes which make up a component are not fetched from a remote, and potentially slow, server unless they are needed for execution.

```
class Hello {
    static boolean __mainChecked = 0;        // 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 4. *The hello world example after it has been processed by our verification service. The vast majority of safety axioms are checked statically. Remaining checks are deferred to execution time as shown. 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 to print a string.*

If any violations are found during the static phase of verification, the service propagates the error to the client by forwarding a replacement class that raises a verification exception during its initialization. Hence, verification errors are handled 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 approach conforms to the specification.

We found that the decomposition of the verifier was instrumental in developing a high assurance implementation. The modular structure simplified manual audits, facilitated automated testing and prevented certain classes of errors.

In contrast, we found that it was hard to characterize the boundaries, configuration, and internal state of the monolithic verifiers found in commercial systems. For example, the precise set of checks performed by the Sun verifier depend on the arguments to the JVM, the class loader used, and the existence of a just-in-time compiler. In one instance, we found that specifying an incorrect length for exception descriptors caused the JVM to crash, even though the verifier contained an explicit check for this case. The crash stemmed from the compiler using the length field before the verifier checks were performed. While a factored architecture does not necessarily result in a modular system, it reduces the likelihood of such errors resulting from inter-component interactions.

4.2. Security

Security for state-of-the-art Java virtual machines [Wallach et al.97] builds on Java's stack-based execution model to express security policies and requires explicit calls from code modules. Access rights are granted by annotating a virtual machine's execution stack. And, they are checked by searching the stack for these annotations. As a result, security for current Java VMs is closely coupled to the virtual machine implementation and to the code executing on it. The security policy is embedded in the code, has no external specification, and, to change a given security policy, all affected code modules have to be rewritten.

Our security services are designed around three principles [Grimm&Bershad97] that address these problems. First, code should be separated from the specification of the corresponding security policy. This separation ensures that a code module's security requirements are clearly documented, can be changed independently of the code, and can be centrally managed. Furthermore, a distinct security policy specification can be mechanically processed, thus permitting high-level reasoning about a system's security behavior. Second, the interpretation of a given policy should be separated from the actual enforcement of that policy. This separation enables the factoring and consequent distribution of system security, and accounts for changing security models and policies. Third, system security should be transparent to an actual code module in the absence of security violations. This transparency simplifies the development of new code modules, and provides a means for imposing security constraints on legacy code.

Consistent with these principles and our overall architecture, we provide system security in a centralized security policy service and a per-machine enforcement manager. The policy service interprets an organization's security policy and, using the binary rewriter, modifies incoming code accordingly. The policy is specified in an XML-based [W3C 98] language, is distinct from code, and reflects an organization's specific security requirements. While XML is somewhat verbose and complex to parse, it is emerging as the lingua franca for Internet-based data representation. Consequently, we expect wide-spread support for XML in editors, browsers and programming environments, making an XML-based policy language a solid foundation for building policy editors and automated analysis tools. The enforcement manager resolves access checks on a client's resources and maintains the corresponding security state. To perform a check, the enforcement manager dynamically queries the policy service for the current security constraints on a resource, thus ensuring that it always sees an up-to-date view of the policy.

Our security specification language is based on two abstractions. It uses security identifiers to represent principals and resources, and permissions to represent the right to perform an operation. Both abstractions also form the basis for the communication protocol between the enforcement manager, which treats them as opaque tokens, and the security policy manager, which interprets them to perform actual policy decisions. The specification language supports three major constructs. The first construct, <namespace>, maps named resources to security identifiers, similar to the name-based security attributes in domain and type enforcement [Badger et al. 95a, Badger et al. 95b]. The second construct, <access-matrix>, specifies legal permissions for pairs of security identifiers [Lampson 71]. It also specifies how to perform transfers between protection domains. The third construct, <class>, specifies where and how to insert calls to the enforcement manager in a code module.

In order to facilitate the exchange of code and security specifications between organizations, our security services use the same data types and representations as the Simple Public Key Infrastructure (SPKI) [Ellison et al. 98]. SPKI is being developed within the Internet Engineering Task Force as a standard for remote authentication and access control based on public keys. Its main contributions are a notion of key-based principals and a simple, yet effective naming system. In our security services, all pertinent representations, such as security identifiers and permissions, are based on SPKI. Furthermore, like SPKI, we use canonical s-expressions [Rivest 97] as the on-the-wire format between the enforcement manager and the security policy service.

Example

To illustrate our security specification language and the operation of our security services, consider the example of restricting file access by applets to only read files under the `/public` directory of the file system. First, we need to

define the appropriate security identifiers (SIDs) and permissions. We use the `applet` SID to represent applet threads, and the `public-file` SID to represent files under the `/public` directory. Furthermore, we use the `fs.read` permission to represent the right to read files.

Next, we need to create a policy specification in our policy specification language. As part of this policy, we need to define a mapping from the file system name space to security identifiers:

```
<namespace name="fs" direction="left-to-right" separator="/">
        <node path="/public" map="incl">
                        <name> public-file </name>
        </node>
</namespace>
```

This specification defines a new namespace, called `fs`, whose names are interpreted from the left to the right and which uses the slash character `/` to separate path components. The namespace has one node with path `/public`. As indicated by the map attribute, all names in the namespace that have this path as a prefix, including the path `/public` itself, map to the `public-file` SID.

Next, we need to grant applets read access to public files. We thus add the `fs.read` tag to the access matrix entry for the SIDs `applet` and `public-file` (not shown). Next, we need to define how and when the enforcement manager is invoked from applets and from the classes that provide file system access. Due to space constraints, we only show this specification for a stripped version of class `java.io.FileInputStream`, which provides read access to files. We omit the specification for other relevant classes, such as `java.io.FileOutputStream` or `java.applet.Applet`. The specification for class `java.io.FileInputStream` and the accordingly rewritten code are as follows:

```
<class name="java.io.FileInputStream">
        <constructor name="FileInputStream(String)">
                <register for="object" from="param" index="0" namespace="fs" />
        </constructor>
        <method name="int read()">
                <check on="object">
                        <tag> fs.read </tag>
                </check>
        </method>
</class>
```

```
public class java.io.FileInputStream {
        public FileInputStream(String name) {
                EnforcementManager.register(this, name, "fs");
                ...
        }
        public int read() {
                EnforcementManager.check(this, "fs.read");
                ...
        }
}
```

The specification requires that two calls to the enforcement manager be injected into class `java.io.FileInputStream`. First, a register operation has to be injected into the constructor. This operation associates the new file input stream object with the SID corresponding to the `String` parameter in the `fs` namespace. Second, an access check has to be inserted into the `read` method. This operation verifies that the current thread has the `fs.read` permission on the current file input stream object.

Now, whenever a new file input stream object is created in a virtual machine, the register operation is executed. The enforcement manager for that VM queries the security policy service for the corresponding SID, providing the `name` argument and the `fs` namespace, and establishes a mapping from the object to the resulting SID. If the file input stream object represents a file under the `/public` directory, it will be associated with the `public-file` SID; if not, it

will be associated with the null SID. On invocation of the `read` method, the access check is executed. The enforcement manager retrieves the SID for the calling thread, which is established at thread creation time and changed on protection domain transfers, and the SID for the file input stream from its security state. It then queries the security policy service for the legal permissions for this pair of SIDs and compares the result with the required permission `fs.read`. If the calling thread is an applet thread and the file input stream object is associated with the `public-file` SID, the legal permissions include the required permission and the operation is complete. If the legal permissions do not include the required permission, the enforcement manager throws a security exception in the form of a `java.lang.SecurityException` object and thus terminates the call to the `read` method.

4.3. Compilation

We are currently in the process of implementing a Java compiler within this architecture. We will report on its design and implementation in the final version of the paper.

4.4. Client Runtime

We have built our own Java runtime to facilitate research with service placement. The runtime implements core services, such as threading, memory management, I/O and synchronization that applications require at execution time. Our implementation provides these services for the PC and Alpha platforms, and relies on the Sun class libraries for the platform APIs. It is structured to permit the integration of virtual machine services when desired. This flexibility allows us to use the same service components under monolithic and distributed configurations. For example, the first three phases of verification can be integrated into the runtime or factored out to a network server. Since the implementation of the fourth phase is varies between architectures, we have two separate implementations. Overall, this structure allows us to evaluate comparable service implementations under different system architectures.

4.5. Performance Monitoring Services

In order to enable virtual machines to monitor their performance and for users to observe application behavior, we implemented a profiling and tracing service. The profiler instruments code to generate a dynamic call-graph [Graham et al. 82]. The tracer instruments applications to generate a timestamped log of all calls to the system libraries [Jones 93]. While we have been using these services extensively for debugging our system, they can also be used for remote monitoring and administration.

4.6. Service Infrastructure

A distributed architecture requires an infrastructure that ties clients together with the services they need. In our current implementation, we adopted an event based filter model for our services. An extended web proxy serves as the I/O substrate for service implementations. The proxy intercepts and processes requests from the clients. An internal filtering API allows logically separate services to be composed together on the same host. The proxy permits a caching filter to be used to avoid fetching and rewriting overhead for shared components. There is also a tracer filter which collects statistics and generates an audit trail from each proxy, and forwards the results to a remote administration console.

The proxy is also responsible for housing the static and dynamic components of VM services. The static components appear to the proxy as rewriting filters which are applied to code passing through the proxy. The dynamic components, which are in essence class libraries that perform additional service functionality, are provided to the clients on demand. Centralization of service state simplifies service upgrades for organizations.

While our service infrastructure requires no changes to the clients, and therefore does not control them directly, a remote management service provides centralized control. At the time a virtual machine fetches its first application, the application's startup code is rewritten to invoke the remote management service. This service then contacts the remote administration console, and a handshake protocol establishes the credentials of the user and assigns an identifier to the session. It is thus possible for a network administrator to remotely examine all hosts in an organization.

5. Evaluation

In this section, we show that our architecture supports secure, manageable and scalable virtual machine services. We first demonstrate that our architecture reduces resource requirements in clients. We then examine the performance overhead of rewriting and conclude that it is negligible compared to typical latencies involved in fetching applications from a wide area network such as the Internet. We show that the rewriting services do not present

bottlenecks for medium-size networks and scale over large numbers of clients. Finally, we compare a monolithic implementation of a verifier to a distributed version, and characterize its performance.

All of the measurements in this section were performed on 200 MHz PentiumPro systems running Windows NT4.0 with 64 MB of memory, connected by a 10Mb/sec Ethernet local area network. Our connection to the Internet is through two 100 Mb/sec links. At the time of the experiments, the links were lightly loaded, with less than 40 Mb/sec peak consumed bandwidth.

	Code size	Data size
Sun JDK	606208	1052072
Netscape JVM	442368	155648
Microsoft JVM	962560	65536
Factored JVM	315392	106496

Table 1. Static code and data sizes, in bytes, for monolithic and factored virtual machines.

The static code and data sizes for different JVM clients are shown in Table 1. The Sun JDK is the reference Java virtual machine (version 1.1.5) from Sun Microsystems. The Netscape and Microsoft JVMs are the versions shipped with the latest browsers by their respective vendors. The measurements for Sun and Netscape exclude the space required for the native AWT implementation, since our factored JVM does not support a windowing system. We could not factor out the size of the AWT libraries from the Microsoft virtual machine because they are bundled into a single library along with the rest of the VM. The measurements demonstrate that a factored architecture requires less resources than a comparable monolithic implementation.

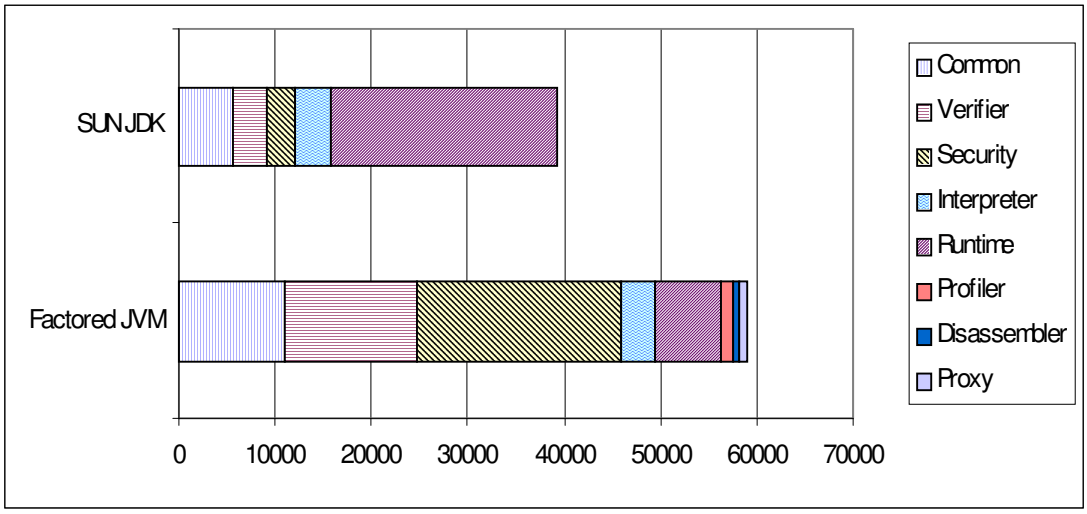
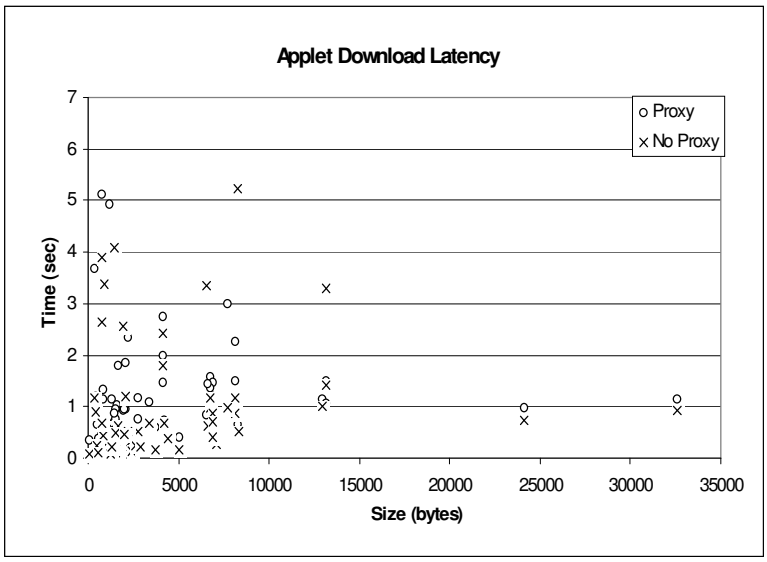


Figure 5. Module lengths of factored and monolithic virtual machines.

Figure 5 evaluates the impact of factoring services on code structure. Services in our factored implementation share a large common code base. Overall, our system consists of 59000 lines of code, whereas the monolithic Sun JDK implementation is comprised of roughly 34000 lines, not counting the native AWT library implementation. We attribute this difference mostly to functionality in our factored virtual machine that the monolithic JVM does not provide. For example, there is no equivalent functionality in the Sun JDK to our proxy, rewriting service, or external security specification.

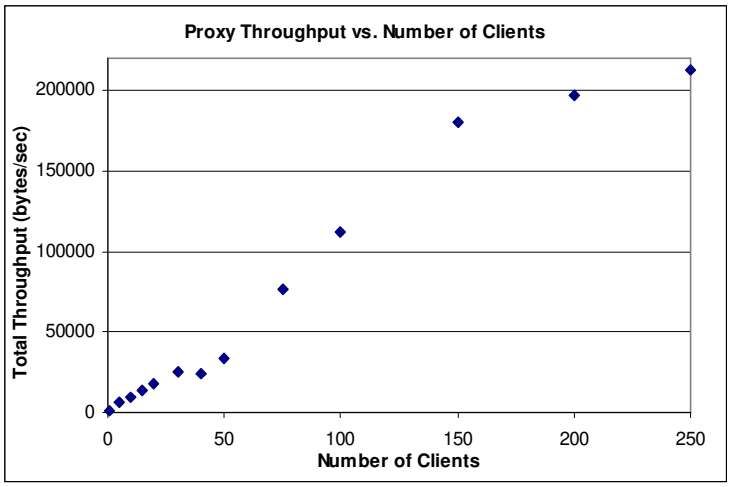
Graph 1 shows the overhead associated with downloading applets through our service infrastructure. We collected a list of all indexed Java applets from the AltaVista search engine, and randomly selected a subset of 75 applets around the Internet. The scatter plot shows the difference between directly fetching these classes from the Internet versus fetching them through our service proxy. Going through our service infrastructure entailed an average delay of 71 milliseconds. This represents a 5% overhead compared to our observed average latency of 1536 milliseconds to download an applet. Given the wide variation, sometimes as high as 15%, we encountered in successive runs of this experiment, the rewriting overhead is small.



Graph 1. Time to download an applet directly from the Internet versus through our service infrastructure.

We then examine the scaling characteristics of the system for large numbers of clients [Graph 2]. We disabled caching in the proxy, and simulated up to 250 clients simultaneously fetching different applets. The graph shows the total proxy throughput as a function of the number of clients. The measurements represent the sustained throughput in our system. Based on these numbers, we conclude that the proxy does not pose a bottleneck for small to medium size networks. In larger installations, the proxy can simply be replicated to handle the increased load.

Finally, we compare our factored verifier to two monolithic verifier implementations. The first is the Sun JDK verifier, which is a highly tuned commercial product. The second is our own verifier integrated into our runtime in a monolithic configuration. We found that, for a large application, namely the Java web server from Sun, verification took 1.218 seconds under the Sun JDK 1.1.5 verifier, 4.190 seconds under our monolithic configuration, and 13.195 seconds in our factored implementation. The factored verifier spends 42% of its time interpreting bytecodes that perform the phase four link checks in Java. We expect this time to be reduced significantly when the bytecodes are compiled. In addition, our system has not yet been optimized.



Graph 2. Sustained throughput of our VM services versus number of clients.

6. Related Work

While virtual machines have evolved extensively since their introduction, their monolithic architecture has remained unchanged. The first commercial virtual machine was the IBM VM system [IBMVM86], which made its commercial debut in 1972. The IBM VM system enabled organizations to run both MVS and CMS on the same physical machine by virtualizing machine resources. As a result of the monolithic structure, all VM services were executed on the same host [Deitel 90]. This structure has influenced many of the virtual machine implementations that followed.

In the early 1970's, virtual machines were also adopted by the language community as a substrate for distributed code. These systems used virtual machines both to retain the portability of applications in an increasingly heterogeneous environment, and to provide high-level abstractions for which compilers could generate code more

easily. P-Code [Psystem] is one such intermediate language that was widely used as part of the UCSD Pascal system. It offered cross-platform portability by targeting a pseudo-machine for compilation, and relying on a runtime interpreter. While the P-System disappeared over time as it did not provide safety guarantees, its portability did influence modern virtual machine design.

Recent virtual machines such as Java and Inferno build on this legacy to provide safety, portability and uniformity in a network computing setting. They rely on a growing set of complex services [Gosling&Yellin 96, Myers&Liskov97, Wallach et al.97] to provide their functionality. We believe that a factored architecture addresses the problems faced by these systems.

The CAGE system from Digitivity takes an alternative approach to factoring services out of clients [Digitivity 97]. CAGE replaces all virtual machines in an organization with a single virtual machine that is physically isolated behind a firewall. All applications execute on this centralized virtual machine, and communicate with clients only for user-interface operations. While this approach has the property that clients are not exposed to untrusted code, it does not address issues of secure sharing of client resources. As a result, while it may be applicable to stateless applets on the web, it does not support applications which require selective access to local resources found on clients.

There are parallels between our work and the use of firewalls for network security [Cheswick&Bellovin94, Mogul et al. 87]. Before the emergence of firewalls, every networked host in an organization had to be protected against all bad packets that it might receive. The rapid acceptance of firewalls demonstrated that it was simpler and more secure to concentrate functionality in a single packet-filter rather than to secure every host in an organization. Modern virtual machines present a similar challenge, except that the services are considerably more complex than packet filtering.

7. Conclusions

We have designed and implemented a new system architecture for network computing based on distributed virtual machines. Our system factors virtual machine services out of clients and locates them in enterprise-wide network servers. The services operate by intercepting application code and modifying it on the fly to provide service functionality. This architecture reduces client resource demands and the size of the trusted computing base, establishes physical isolation between virtual machine services and creates a single point of administration. Such a distributed virtual machine architecture can provide substantially better integrity and manageability than a monolithic architecture, scales well with increasing numbers of clients, and has comparably low overhead.

8. Acknowledgements

We would like to thank Jim Roskind of Netscape, Charles Fitzgerald and Kory Srock of Microsoft, and Tim Lindholm, Sheng Liang, Marianne Mueller and Li Gong of Javasoft for discussing the internals of their systems with us. Sean McDirmid implemented parts of our verifier. Bibek Pandey collected the list of applets that were used in our performance evaluation.

9. References

[Accetta et al. 89] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid R. F., Tevanian, A. and Young, M. W. “Mach: A New Kernel Foundation for Unix Development.” USENIX 1989.

[Adl-Tabatabai et al. 96] Adl-Tabatabai, A., Langdale, G., Lucco, S. and Wahbe, R. “Efficient and Language-Independent Mobile Programs." In *Conference on Programming Language Design and Implementation*, May, 1996, p. 127-136.

[Badger et al. 95a] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Pratical Domain and Type Enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66-77, Oakland, California, May 1995.

[Badger et al. 95b] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A Domain and Type Enforcement UNIX Prototype. *In Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 127-140, Salt Lake City, Utah, June 1995.

[Bershad et al. 95] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C. “Extensibility, Safety and Performance in the SPIN Operating System.” In *Proceedings of the Symposium on Operating System Principles*, 1995.

[RFC1157] Case, J., Fedor, M., Schoffstall, M. and J. Davin, "Simple Network Management Protocol", RFC 1157, May 1990.

[Cheswick&Bellovin94] Cheswick, W. R. and Bellovin, S. Firewalls and Internet Security: Repelling the Wily Hacker. Addison-Wesley, 1994.

[Custer 93] Custer, H. Inside Windows NT. Microsoft Press, 1993.

[Dean et al. 97] Dean, D., Felten, E. W., Wallach, D. S. and Belfanz, D. "Java Security: Web Browers and Beyond." In Internet Beseiged: Countering Cyberspace Scofflaws, D. E. Denning and P. J. Denning, eds. ACM Press, October 1997.

[Deitel 90] Deitel, H. M. An introduction to Operating Systems. Addison-Wesley, Reading, MA 1990.

[Digitivity 97] Digitivity, Inc. "Digitivity Introduces the First Internet Applet Management System for the Secure Deployment of JAVA." http://www.digitivity.com/press_062397.html, June 1997.

[Ellison et al. 98] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, T. Ylonen. Simple Public Key Infrastructure. Internet Draft, Internet Engineering Task Force, March 1998.

[Engler et al. 95] Engler, D. R., Kaashoek, M. F. and O'Toole, J. "Exokernel: An Operating System Architecture for Application-Level Resource Management." *In Proceedings of the Symposium on Operating System Principles*, 1995.

[Freund&Mitchell 98] Freund, S. N. and Mitchell, J. C. "A type system for object initialization in the Java Bytecode Language." To appear in *ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, 1998.

[Gosling&Yellin 96] Gosling, J. and Yellin, F. The Java Application Programming Interface, Volumes 1 & 2. Addison-Wesley, 1996.

[Graham et al. 82] Graham, S.L., Kessler, P.B. and McKusick, M.K. "gprof: A Call Graph Execution Profiler." In Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No. 6, pp. 120-126, June 1982.

[Grimm&Bershad97] Grimm, R. and Bershad, B. "Providing Policy-Neutral and Transparent Access Control in Extensible Systems." Technical Report UW-CSE-98-02-02, February 1998.

[IBMVM86] IBM Corporation. Virtual Machine/System Product Application Development Guide, Release 5. Endicott, New York, 1986.

[Inferno] Lucent Technologies. Inferno. http://inferno.bell-labs.com/inferno/

[Jones 93] Jones, M. B. "Interposition Agents: Transparently Interposing User Code at the System Call." In Proceedings of the Symposium on Operating System Principles, December 1993, pp. 80-93.

[Lampson 71] B. W. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437-443, Princeton, New Jersey, March 1971. Reprinted in Operating Systems Review, 8(1):18-24, January 1974.

[Lindholm&Yellin96] Lindholm, T. and Yellin, F. The Java Virtual Machine Specification. Addison-Wesley, 1996.

[Madany 96] Madany, P. "JavaOS – Java on the Bare Metal." JavaOne 1996.

[Mogul et al. 87] Mogul, J. C., Rashid, R. F. and Accetta, M. J. "The Packet Filter: An Efficient Mechanism for User-level Network Code." In *Proceedings of the Symposium on Operating System Principles*, November 1987, pp. 39-51.

[Myers&Liskov97] Myers, A. C. and Liskov, B. "A Decentralized Model for Information Flow Control." In *Proceedings of the Symposium on Operating System Principles*, 1997

[Saltzer&Schroeder 75] Saltzer, J. H. and Schroeder, M. D. "The Protection of Information in Computer Systems." In *Proceedings of the IEEE*, 63(9):1278-1308, September 1975.

[Rivest 97] R. Rivest. S-Expressions. Internet Draft, Internet Engineering Task Force, May 1997.

[Sirer et al. 98] Sirer, E. G., Grimm, R., Bershad, B. N., Gregory, A. J. and McDirmid, S. "Distributed Virtual Machines: A System Architecture for Network Computing." European SIGOPS, September 1998.

[Spyglass 94] Software Development Interface for Dynamic Data Exchange. http://www.spyglass.com/products/smosaic/sdi/sdi_spec.html

[Stata&Abadi 98] Stata, R. and Abadi, M. "A type system for Java bytecode subroutines." In *Proceedings of the 25th Symposium on Principles of Programming Languages*, January 1998, p. 149--160

[Tanenbaum et al. 90] Tanenbaum, A.S., Renesse, R. van, Staveren, H. van., Sharp, G.J., Mullender, S.J., Jansen, A.J., and Rossum, G. van: "Experiences with the Amoeba Distributed Operating System," Commun. ACM, vol. 33, pp. 46-63, Dec. 1990

[Thompson 84] K. Thompson. Reflections on Trusting Trust. Communication of the ACM, Vol. 27, No. 8, August 1984, pp. 761-763.

[Psystem] University of California, Irvine. p-System: Description, Background, Utilities. http://www.ics.uci.edu/~archive/documentation/p-system/p-system.html

[W3C 98] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. Edited by T. Bray, J. Paoli, C. M. Sperberg-McQueen, February 1998.

[Wallach et al.97] Wallach, D. S., Balfanz, D., Dean, D. and Felten, E. W. "Extensible Security Architectures for Java." In *Proceedings of the Symposium on Operating System Principles*, 1997.

[Walker et al.83] Walker, B., Popek, G., English, R., Kline, C. and Thiel, G. "The LOCUS Distributed Operating System." In *Proceedings of the Symposium on Operating System Principles*, 1983, p. 49-69.