Implementing Network File System Policies with FileWall

Stephen Smaldone, Aniruddha Bohra, and Liviu Iftode
Department of Computer Science, Rutgers University
110 Frelinghuysen Road, Piscataway, NJ 08854, USA
{*smaldone,bohra,iftode*}*@cs.rutgers.edu*

ABSTRACT

Managing network file systems in large deployments is a critical challenge facing administrators today. Network file systems are widely used, are standardized, and provide acceptable performance. These systems are designed for the *least common denominator* of functionality, across all deployments to enable widespread use across diverse client systems. Unfortunately, specific deployment scenarios require different *policies* that govern file system access. The rigid structure of current network file systems makes modifying policies and mechanisms equally difficult.

In this paper, we present a novel approach to implement network file system policies through message transformation, external to clients and servers. We present FileWall, a file system proxy, through which administrators can easily extend network file system policies for monitoring, access control, maintenance, and semantic extensions. We also present a policy specification language, FWL, which allows administrators to specify policies in a few lines, without being encumbered with implementation details.

We have implemented FileWall using the Click modular router framework for the NFS protocol, and present solutions for four real-world administrative problems using our system — maintaining per-client file system statistics, implementing temporal access control, improving file handle security, and supporting client transparent failover. Through our evaluation, we show that FileWall imposes minimal delays, the interposition overheads are low, and the performance of FileWall is comparable to a simple network tunnel.

1. INTRODUCTION

With the unprecedented growth of storage infrastructures in size and number of users, network file system management is a critical challenge facing administrators today. Network file systems are designed for the least common denominator of functionality, across all deployments, to enable widespread use over diverse client populations. While performance enhancements and protocol optimizations for homogeneous data access mechanisms benefit all users of a network file system, management functionality varies from one deployment to the other. As a result, administrators are forced to implement custom solutions for their individual local installations. Unfortunately, integrated data access mechanisms and file system policies, which are common in network file systems today, make it as difficult to modify policies as the data access mechanism.

We propose a novel approach to implement network file system policies *externally*, without modifying either the client or the server, by transforming the file system messages flowing between them. There are several advantages of using our approach. First, network context is always available in the message streams and by implementing policies at a single point in the network allows immediate and simultaneous enforcement of policies across all clients. Second, protocols are standardized and are independent of the host file system implementations. Implementing policies by transforming messages while maintaining protocol invariants allows portable and easy to specify policies. In contrast, implementing policies at either the client or the server requires the policy writer to understand various OS-specific details. Third, by implementing policies external to the servers and clients, we can handle client-transparent failovers and server maintenance without shared storage, which is not possible otherwise. Finally, the separation of concerns of *network* aware policy enforcement and the file systems, allows them to evolve independently.

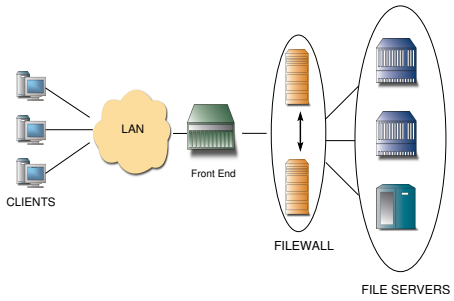

Figure 1: FileWall architecture.

We do not advocate *all* file system policies be implemented through message transformation. Policies that interact closely with the local file system and do not require network specific knowledge, for example data transformations through encryption or content addressable storage, versioning, etc. are best implemented by extending the local file systems. Several ideas have been proposed and frameworks for extending *local* file system policies have been implemented. Recently, stackable file systems [30] have provided a platform and a language for easily extending file systems to implement a spectrum of policies e.g. intrusion detection, file system tracing, fine-grained access control, data and metadata versioning, etc. While local file system policies can easily be implemented and extended, implementing network file system policies remains a challenge for the administrators.

In this paper, we present FileWall, a proxy that sits in the network path between the clients and the servers, and transforms file system messages to implement policies, without modifying either the client or the server. Figure 1 shows FileWall architecture. The file servers and FileWall are trusted and reside in the same network domain, whereas the clients are external. Administrators can program the FileWall to implement enterprise-wide policies through protocol message transformation. FileWall provides a message transformation framework, persistent state storage, and a high-level language that allows easy policy specification for administrators.

At the highest level, FileWall captures all network file system messages, parses each message to extract its file system attributes, and evaluates policies by transforming these attributes and reinjecting the transformed messages into the network. New request messages can be generated at FileWall to retrieve additional attributes from the file system and new response messages can be generated to respond directly to the client, *without* consulting the file server. Policy specific state can be maintained either in-memory or using persistent storage. Policies are implemented by transforming message attributes using the stored state as the *access context*, and reinjecting the transformed messages into the network.

We identify four classes of policies that define the desiderata for the FileWall framework: *(i)* Monitoring policies, which utilize the network context to provide a monitoring framework for administrators. For example, a policy that maintains per-client, per-user file system statistics that are not available in existing network file system implementations. *(ii)* Access policies, which extend the traditional rwx access control framework provided by network file systems. For example, a policy to implement fine grained access control, which includes not only users and groups of users, but also the identity of the client systems to arbitrate access to the file system. *(iii)* Maintenance policies, which include security, client transparent failover, admission control, and extension policies that use protocol invariants to provide enhanced functionality. For example, a policy that restricts object identifiers (file handles) to those that have been seen in the server to client message stream, prohibiting a client from passing arbitrary identifiers to the server. *(iv)* Semantic policies, which extend the file system interface or modify the semantics of network file system access. For example, a policy that implements semantic file systems [24], which translate a virtual name to a query that generates data using results from multiple file system requests. This classification is not absolute but provides the necessary model to design FileWall.

We define a high-level policy specification language, FWL, which provides support to the policy writer (administrator), for most commonly performed message inspection and transformation tasks. At the same time, we allow a skilled administrator the ability to access the raw message stream directly and implement complex policies. The FileWall compiler translates the high level specification to a native programming language, C or C++. The resulting policy is then compiled to object code by the default compiler.

There are two main features that separate FWL from similar packet filtering and stackable file system languages. We provide an *extraction operator*, which parses a message into independent components based on the protocol specification, and state maintenance through *persistent associative arrays*, which allow a policy writer to store its state on persistent storage.

FileWall is implemented using the Click modular router [16]. It is portable as it relies on Click and the OS, only for capturing network file system packets and injecting them into the network. In this paper, we describe a user-level implementation of FileWall. We demonstrate that even at the user-level, FileWall does not impose significant overheads on client-perceived performance. At the same time, even these overheads can be reduced through in-kernel or hardware based implementations of FileWall. To realize such implementations, only the FWL compiler must be modified to generate OS specific code, the policy writer need not rewrite the policies.

We have implemented several network file system policies with FileWall. We describe our experience with four policies in this paper: a stateless statistics maintenance policy, a temporal access control policy, a file handle security policy, and a client transparent failover policy. In our experience, policy implementation with FileWall significantly reduces development time, as policies can be specified in a few lines without any knowledge of the end-host implementation.

We evaluate our system using microbenchmarks, latency sensitive workloads, and real-world workloads. Our results demonstrate that FileWall imposes small interposition overheads of less than $40\mu s$ and these overheads increase linearly with the number of policies executed by FileWall. We also demonstrate overheads of less than 15% for the real-world workloads compared to the default NFS for the same setup.

This paper is organized as follows. Section 2 discusses the motivation for FileWall. Section 3 describes the FileWall model. Section 4 presents the FileWall design. Section 5 describes the FileWall policy specification language. We describe our example policies in Section 6 and present the implementation in Section 7 and the evaluation of our system in Section 8. Section 9 summarizes the related work and Section 10 concludes the paper.

2. MOTIVATION

Network file system administrators struggle to implement new file system policies. Policies that are easy to state require significant effort on the part of the administrators, either as implementation and configuration overhead, or for deploying the modified policy across the system. In the following, we list some examples that demonstrate the real-world scenarios where such policies are desirable. Without FileWall, implementing each of the following policies in existing systems, would require understanding and modifying the OS implementation of the server and in some cases for all clients. In Section 6, we will present implementations of these real-world policies using FileWall.

- *Statistics (Section 6.1):* An important tool for administrators to understand the performance of network file systems is the ability to collect statistics about the messages flowing between the clients and servers. Today, these statistics are collected by the clients and servers independently, and are extracted via utilities residing at each system (e.g., nfstat). This is sufficient when an administrator is only interested in the aggregate statistics for a file server or for a particular client. For large deployments of many clients accessing multiple servers, these statistics are inadequate, as these utilities do not provide per-client/per-server statistics, which would be most useful in these large deployment scenarios. Administrators would like to implement policies that enable fine-grained monitoring and statistics maintenance.

- *Access Control (Section 6.2):* File management has traditionally been split between file owners and file system administrators. Owners typically manage file organization, naming, and protection, while administrators control file system configuration, backup and recovery, disk space management, etc. File access policies based on file attributes, e.g., permissions, access control lists, etc., are simplistic, static, and their control is at the discretion of the file owner. From the administrator's perspective, file access policies are difficult, if not impossible, to impose. Administrators have no simple way to define complex file access policies based on dynamic access information such as number of accesses, time, location, etc. For example, a time-based access policy, which escalates a user's rights for a specified period of time, afterward revoking the additional rights. While it is easy to grant access, a time-based revocation cannot be performed without human intervention.

- *Bugfixes (Section 6.3):* A simple survey of the Linux NFS bug database reveals several instances where the server was crashed or otherwise compromised by malformed messages containing badly constructed file handles, large buffers, invalid credentials, etc [14, 15]. These problems arise due to the expectation of the server that all clients strictly follow the file system protocol and insufficient error checking at the server. Administrators would like to implement policies that strictly enforce the protocol invariants to prevent malformed messages from reaching the server.

  For example, an NFS server provides clients a unique and persistent file handle for each file it serves. As described in [27], these file handles reveal information about the server, such as OS version and date of file system creation. Moreover, guessing file handles is easy, since they are deterministically created, and allows for an attacker to bypass mount based authentication and parameter negotiation. Administrators would like to implement a policy which protects servers against such file handle based attacks.

- *Client Transparent Failover (Section 6.4):* File server replication is a well known technique for providing high availability to clients. Typically, one or more backup servers is maintained as replicas of a primary server. When the primary server fails or is brought down for maintenance, one of the backup servers is promoted to replace the primary and continues to service requests for clients. For this to be transparent to clients, network reconfiguration is required (e.g, DNS redirection, IP takeover, etc.) to redirect client requests to the new primary server. Finally, clients must remount existing network file systems to refresh file handles. Administrators would like to implement policies that support client transparent failover which does not require clients to remount and avoids all downtime.

3. FileWall MODEL

FileWall is a file system proxy that intercepts all messages between the client and the server, and transforms these messages to implement policies. FileWall is collocated in the same network segment as the servers. In our model, all file system accesses are remote. That is, clients cannot log on directly to the server and modify the file system state locally. Therefore, all modifications to the file system state are performed over the network and pass through FileWall.

FileWall is transparent to clients and is trusted by the file servers. It shares the server's authentication keys, which enables FileWall to intercept, interpret, and transform encrypted or signed messages.

File system policies are defined by administrators who have exclusive access to FileWall. A single policy represents a unit of FileWall processing. Similar to an object oriented program, each policy is defined by a class that determines its behavior. Policies adhere to a common interface and are connected in a chain. FileWall mediates the transfer of messages between policies and schedules the policies. FileWall provides an interface for administrators to dynamically update the policies.

FileWall maintains state in a per-policy state store called *access context*. Policy writers can choose to store the access context in memory or on persistent storage. The access context is composed of environment variables, e.g. time and current resource usage, and temporary or persistent state generated by the policy over its execution.

Policies transform file system messages using the access context as they flow through the chain. Message transformation includes

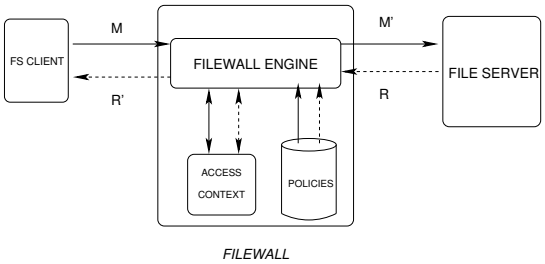

Figure 2: FileWall Architecture. The solid lines show the path of the request while the dotted lines show the path of the response. FileWall transforms the request from M to M', and the response from R to R'.

modification of a message, new request messages generated on behalf of the client, and new response messages generated on behalf of the server. The access context is also updated during execution.

Figure 2 shows the flow of a client request message (M) and the response message (R) through FileWall. The client sends its request to the server. The FileWall engine intercepts this message and invokes policies that use the access context to transform this message to M' and forwards M' to the server. The server responds with a message (R) which is transformed by FileWall to R' and sent to the client, which receives the transformed message as the file system response.

3.1 Message Transformation

A file system message defines a single unit of transfer between clients and servers, which is composed of one or more packets sent over the network. For a datagram protocol, e.g., UDP, each packet contains a file system message. For stream oriented protocols, e.g., TCP, client connections are made directly to FileWall, which in turn establishes connections to the servers, and messages are identified by special record markers. Each message contains attributes that determine the file system operation. A message sent by a client contains the attributes corresponding to a remote procedure call (RPC), while a server sends the response to this call.

FileWall is responsible for extracting file system attributes from messages and reconstructing messages with new or transformed attributes. Therefore, it must understand the data representation used by the protocol. In this paper, we describe FileWall built for the ONC/RPC protocol that uses the XDR data representation. However, FileWall can be easily extended to work with other data representation standards.

Policies can choose to modify one or more attributes in a message. Additionally, on receiving a request message, policies may respond without sending the original message to the server. For example, in an access control policy, FileWall may generate an access denied message as a server response. Finally, if FileWall needs additional information from the server, it generates new request messages, which are sent to the server. Responses to these messages are not forwarded to the client. For example, a replication policy must send the request message to all the replicas while forwarding exactly one response to the client.

FileWall operates on file systems that follow a transactional mode, that is every file system operation initiated by the client must receive a response from the server. The client state is not updated until the server replies with a successful response. Most existing network file systems, for example, NFS and CIFS, follow the client driven transactional model.

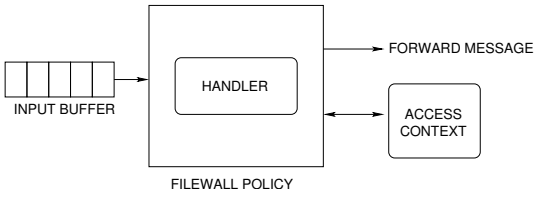

Figure 3: FileWall policy architecture.

3.2 Access Context

Policies are evaluated using the attributes contained in the file system message, in context of the state maintained by the policy or present in the execution environment. We call this state the *Access Context*. There are two components of the access context: *(i)* Static context, and *(ii)* Dynamic context. Each of these components of the access context are available to policy writers.

Static context is read-only state specified by the policy writer during initialization. This state is not updated during execution. Examples of such state include lists of user identifiers, list of file servers, and servers' authentication keys.

Dynamic context contains state generated by policies during execution. Policies can store arbitrary state in the dynamic context and retrieve it at a later stage. FileWall does not interpret the state, but treats it as a black box. Policies may choose to store their dynamic context in persistent storage through a database, which guarantees the ACID properties (Atomicity, Consistency, Integrity, and Durability) of the stored state. By default, dynamic context stored in memory is purged on every restart and must be rebuilt from scratch.

3.3 Policies

A policy is a unit of FileWall processing. Its behavior is defined by handlers that are invoked on receiving messages. Figure 3 shows a FileWall policy. The arrows represent the policy interaction with FileWall through function calls. FileWall manages fixed-sized input buffers for a policy. FileWall schedules a policy by calling the policy handler for a message stored in the buffers. A policy can interact with FileWall through either reading/writing the access context, or by forwarding a message, discarding a message, or creating a new message. FileWall places these message in the input buffer of the next policy in the chain.

During execution, FileWall ensures that each policy is invoked exactly once for a request and once for the corresponding response message (Section 4.4). Policies use the attributes contained in messages and the state maintained in the access context for evaluation. Policies maintain state in the dynamic context component of the access context and update it when invoked to handle messages.

To guide the design of FileWall mechanisms, we first classify the policies implemented with FileWall based on their goals. Such classification allows us to reason about common tasks performed by general *classes* of policies instead of a specific instance. Classifying policies based on high-level goals also identifies a set of guidelines for policy writers to develop new policies for FileWall.

We define four classes of policies: *(i)* Monitoring Policies, *(ii)* Access Policies, *(iii)* Maintenance Policies, and *(iv)* Semantic Policies. In the following, we present a brief overview of each of the policy classes while focusing on the message transformations and the access context of the class. Table 1 presents a summary of the policy classes and their characteristics. In the table, the rows show the policy classes, while the columns record a feature. A ✓ in a cell indicates an FileWall feature essential to the policy class, while a × indicates a feature that is not used by this class. A *Policy* in a cell indicates that this feature is used in some instances of the policy class. Access control policies either forward both request and response messages, or generate a new *Deny* message as a response.

Monitoring Policies: These policies passively observe the message streams and do not interfere in the client-server communication. The goal of this class of policies is to extract information of interest from the message stream and use this information in external monitoring applications.

Message transformation is not performed by these policies. The dynamic context is used to record the monitoring log or history of accesses and is update only. The static context is not used and no policy decisions depend on the environment.

Instances of this class include policies that record statistics (e.g., per-client, per-user operation counts and latencies), file system tracing policies, which record all messages in a log that can be used to analyze access patterns or replay traces for performance evaluation, etc.

Access Policies: These policies arbitrate the flow of messages from the client to the server based on the access control logic. These policies make a Boolean decision — to allow or deny an incoming client request message from reaching a server. If the decision is to allow the request, the message is forwarded to the server unmodified. For requests that are denied, the policy generates a new request denied message, which is sent to the client without forwarding the request to the server.

Message transformation in this class of policies is limited to generating *Deny* messages and no attributes are transformed in either the request or the response. The dynamic context is used by specific instances of policies that base their decisions on the access history or the server response latency. The static context is used for configuration, e.g., to populate the user identifier and ACL databases.

Instances of this class include fine-grained and temporal access control policies, which allow or deny requests based on the ACL configuration, network identity, and current access time. This class includes admission control policies, which allow or deny requests based on the server load determined by the number of pending requests, server response latencies, etc.

Maintenance Policies: These policies are designed to *virtualize* file servers and allow administrators to perform client-transparent, online server maintenance. These policies use *indirection* in mapping the physical or server generated object identifiers (file handles) to virtual or client visible file handles. This indirection allows policies to dynamically map requests to the appropriate server identifiers.

Attribute transformation is an essential primitive for this class of policies. The dynamic context maintains the virtual to physical mapping of attributes. The static context is used for policy configuration. These policies use the environment to control the policy behavior.

Instances of this class include security policies that prevent clients from determining server information from file handles by generating a per-client random file handle. This policy enforces the invariant that clients cannot generate file handles. Other policies include client transparent failover, where server replicas are maintained by generating requests to multiple servers and responding to the client using the identity of the currently designated primary replica.

Semantic Policies: These policies extend existing network file systems by implementing novel semantics. This class of policies is broad. It includes all policies that do not fall into one of the above classes. Moreover, policy writers have complete control over message transformation as well as the access context.

Depending on the specific instance, policies of this class may transform messages or generate new messages. Policy decisions

Class	Dynamic Context	Message Transformation			
		Req.Attr.	Rsp. Attr.	New Req	New Rsp
Monitoring	Logs	×	×	×	×
Access	ACL Tables	×	×	×	Deny
Maintenance	Attribute maps	✓	✓	Policy	Policy
Semantic	Policy	Policy	Policy	Policy	Policy

Table 1: FileWall Policy Classification

are left to the policy writer. Since new semantics are implemented by these policies, the client view as well as the interface to the the file system may be modified.

Instances of this class include policies that implement consistency and atomicity semantics beyond those supported by the underlying file system. These policies can implement update consistency across multiple files or directories, support multiple versions of the same file, etc. Other examples of policies in this class are those that reorganize files based on access history [26].

4. FileWall DESIGN

In this section, we present the design of the FileWall system. First, we describe the guidelines we follow while designing the FileWall mechanisms, followed by a description of the basic building blocks of our system.

4.1 Design Guidelines

We identify the following guidelines for FileWall design:

- *Minimalism:* Our goal while designing FileWall was to maintain extensibility and portability across file system protocols and operating system platforms. To achieve these goals, we chose to keep the core of the FileWall system small and provide a set of policies that can be used as basic building blocks to implement functionality external to core.

  The minimal functionality of the FileWall core leads to a small, reliable, and easy to maintain code-base. Moreover, by providing a small kernel of functionality, we place minimal restrictions on policy construction and specification making it a suitable platform for future extensions.

- *Lazy Operation:* FileWall design makes no assumptions about what services are required by a policy. We delay all computation on behalf of policies until the last moment possible. For example, FileWall does not extract attributes until a policy actually requests specific attributes. Attribute extraction is incremental, and stops as soon as the requested attribute is available. In contrast, an eager FileWall would extract all attributes on receiving messages and invoke policies with all attributes available.

  Lazy operation has two advantages. First, it avoids unnecessary computation by FileWall, especially for policies that use few attributes, e.g., a statistics monitoring policy that uses only file system call and transaction identifiers. Second, lazy operation enables accurate resource accounting as FileWall performs all tasks in context of a policy.

- *Soft-State:* FileWall introduces an additional element in the client-server path and its failure would lead to an otherwise healthy server being unavailable to clients for service. Therefore, at all stages, our goal was to maintain only *soft* state in FileWall.

  The FileWall core maintains only soft-state. Policies that require hard state, must explicitly store such state and retrieve it on a restart. FileWall treats such state as opaque data and stores it in a database, which provides ACID properties. However, since database accesses might be expensive, policy writers must try to limit its usage for state components that cannot be reconstructed during operation.

4.2 Policy Execution

Listing 1 shows an example FileWall policy that simply forwards all messages it receives without any processing. The parent class, **FWPolicy**, is the parent of all policies in our system. A policy class must define its constructor and destructor functions, and must provide two accessor functions, `name` and `id`, shown in the example. During initialization, the policy calls the FileWall function `getid()` to get a system-wide unique identifier.

As shown in the example, a policy must also define two handlers: the request and the response handler. `RPCMsg` is the class that encapsulates the file system message. It contains the raw packet buffers received over the network, the stream representation (XDR representation) of these buffers, and any attributes that have previously been extracted from this stream. For response messages, the RPCMsg class includes the corresponding request buffers as well. In the example, no attributes are extracted and the FileWall `forward()` function is called.

The FileWall scheduler (Section 4.5) picks the next policy. The scheduler then dequeues a message from the head of the input buffer and calls the policy handler. All processing in the policy uses this message and the access context. For response messages, the associated request message is also provided to the handlers. The request and response messages are associated using the unique transaction identifier represented by the two tuple, $< FlowID, XID >$, where $FlowID$ is the client's unique network identifier and the XID is the RPC transaction identifier.

A FileWall policy is assigned a unique identifier, is independently scheduled and runs to completion. Every execution of a policy is in context of a message. Messages are processed in order.

```
class NullPolicy : public FWPolicy {
  public:
    NullPolicy () {}
    ~NullPolicy () { }
    const char *name() { return ``NullPolicy''; }
    void initialize (int id, const char *config) { _id = id; }
    void req_handler (RPCMsg *msg) { forward(msg); }
    void rsp_handler (RPCMsg *msg) { forward(msg); }
  private:
    int _id;
};
```

Listing 1: A NULL policy that passes through all requests and responses

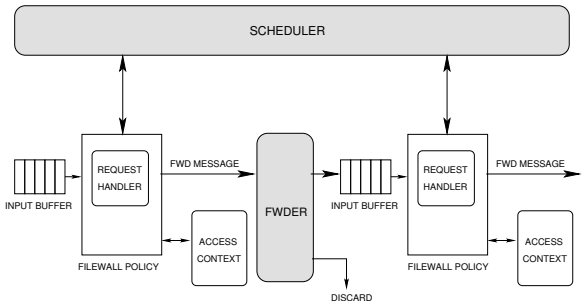

Figure 4: Policy chains with the FileWall scheduler and forwarder. The policies are identical to Figure 3

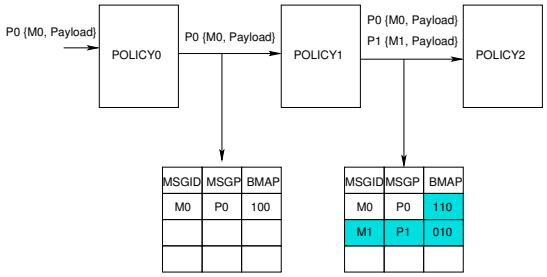

Figure 5: Pending Request Map Example.

FileWall maintains a policy-specific access context private to each policy. FileWall also maintains the *pending request map*, a hash table, which store all pending requests, and records the policies that have handled a message. We show in Section 4.4 how the pending request map is used to match requests and responses, and enforce exactly once semantics in a chain of policies.

FileWall defines a mechanism for policies to return unmodified, transformed, or newly created messages after processing. We call this interface the *forward* interface, where the message to be sent out of the policy is forwarded to FileWall. If a policy must save a message without forwarding it, it must store it in its private access context and return without forwarding any messages.

4.3 Policy Chains

FileWall policies can be linked as shown in Figure 4. Each element of the chain is a policy shown in Figure 3. Policy chains are defined by the administrator using a plain-text configuration file that is parsed by FileWall. This file lists the policies in the order they must see client request messages. Policies are identified by name and the same policy cannot be included in the chain twice.

Two FileWall components, *Scheduler* and *Forwarder* (shaded boxes in Figure 4), enable policy chains. The forwarder is invoked by policies through the `forward()` function call. On receiving a message, the forwarder first updates its state. It then classifies the message as a request or a response and finds the next policy and its buffers in the global policy chain. Finally, it places a reference to the pushed message in the appropriate buffer of this policy. If this buffer has no empty slots, the message is dropped. Dropped messages are no different than the network dropping the packets and do not affect the correctness of the protocol.

Following the principle of minimalism, FileWall takes advantage of policy chaining and defines two default policies *SEND Policy* and *RECV Policy* to send and receive messages respectively, *external* to the FileWall core. These policies are placed by default at the end and the beginning of the policy chains. Therefore, the next processing stage can be a policy or the end of the chain, where the message is sent out over the network.

Policies are organized as two ordered queues: the input and the output queue that are mirror images of each other. That is, the policy at the head of the input queue is at the tail of the output queue. Intuitively, if a policy, P0, sees a request R0 before P1, then the response for R0 will be processed first by P1 and then by P0. The start (RECV) and the end (SEND) of the policy queues are fixed and receive and send messages respectively. A message that exits FileWall is guaranteed to have been processed by each policy in the predefined order.

4.4 Matching Requests and Responses

FileWall maintains a global pending request map, which is initialized when *request* messages are generated, either by receiving them from the network, or by policy handlers. The pending request map maintains the messages indexed by the unique transaction identifier, and a bitmap of all policies that have handled the message.

Figure 5 shows the flow of two messages through a policy chain. There are three policies in the system with identifiers 0, 1, and 2. The message P0 has the message identifier M0 and is seen by all three policies. As each policy forwards the message, the bit in the pending request map corresponding to this policy is set. Policy 1 generates a new message, which has not been seen by Policy 0. On creation, FileWall allocates a new entry in the pending request map and sets the bit for policy 1. The modifications to the request map are shown as shaded boxes.

On receiving a response message, the corresponding request message is identified, is retrieved from the pending request map, and is presented to the policy handler along with the response message. Before handling the response, FileWall checks if its bit is set in the bitmap. If not, the message is passed to the next policy in the chain. If none of the successor policies in the response queue handle the message, the message is discarded. Following the principle of lazy execution, the messages are evaluated for handling just before execution.

FileWall ensures the exactly once execution semantics through the bitmap. Response messages are handled only by policies that have previously handled the corresponding request message. Therefore, if a policy generates a new request message, none of its predecessors in the policy chain see the response. Similarly, if a response message is generated by a policy, the successors in the policy chain see neither the request nor the response message.

If the message is dropped, either by a policy or by the network, the pending requests occupy FileWall memory. FileWall defines a timeout that purges the request messages and clears the bitmap. All responses for this request are subsequently discarded.

4.5 Policy Scheduling

FileWall gains control of execution either asynchronously, on receiving a new packet, or synchronously, on completion of a policy handler. On receiving a new packet, FileWall simply inserts it in the input buffer of the *RECV* policy. Note that the packets are well formed file system packets. Reassembly and classification of packets are performed external to FileWall. No scheduling decisions are made during this execution.

FileWall schedules the policies synchronously by invoking the policy handler. Policies are scheduled round-robin, from left-to-right in the chain and run to completion. More complicated schedul-

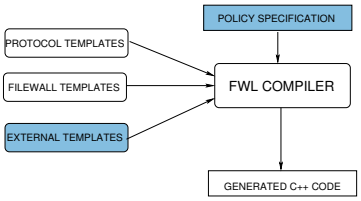

Figure 6: FWL code generation

ing algorithms, for example proportional fair share or lottery scheduling, can be used as replacements for our default implementation.

5. POLICY SPECIFICATION LANGUAGE

We have designed a high-level language for policy specification, FWL, which allows administrators to specify complicated policies without encumbering them with repetitive tasks. FWL programs are compiled into a native language (C or C++) using protocol specific templates. Figure 6 shows the FileWall policy compilation procedure. The policy writer must specify the external templates and the FWL policy specification, which are shown as shaded boxes in the figure. The FileWall system defines the protocol and FileWall templates. The generated code is then compiled into object code using the native (C or C++) compiler.

In addition to the protocol templates, we define FileWall templates that provide the necessary glue code to create working native language code for a policy. Finally, the policy writers can specify implementations of external functions, which are declared in the policy specification and are invoked whenever a statement in the policy accesses the function.

5.1 FWL Program Structure

A FWL program consists of *sections* that correspond to the respective processing in the policy. We define three sections: Configuration, Request, and Response that handle the corresponding FileWall processing. The configuration section gets translated as class variables, the request section is placed in the body of the request handler, and the response section is placed in the response handler for the policy. A FWL policy specification has the form:

```
@CONFIG::ClassName {
  NAME = ``EXAMPLE'';
  /* Declarations, Configuration variables */
}
@REQ {
  /* Request handling code */
}
@RSP {
  /* Response handling code */
}
```

The FileWall templates are skeleton class definitions, which are filled in using the FWL specification. Special markers in the template files indicate the beginning and the end of the generated code block. The string following the @CONFIG preamble is chosen as the name of the generated policy class.

The configuration section defines the global or class private variables. The FWL compiler identifies the type of these variables either through type inference or through explicit declaration.

In addition to variables, policy writers can defined external functions in the configuration section. These functions must be specified in the external templates. If the external functions are not defined by the policy writer, a link-error is generated by the C++ compiler.

```
nfs_fh3 *extract_FHREQ(msg *msg) {
  nfs_fh3 *fh = NULL;
  rpc_msg *rpcm = (rpc_msg *)msg->reqpkt;
  /* Get the operation */
  uint32_t op = rpcm->rm_call.cb_proc;
  ...
  /* Remove the cred and verf structures */
  char *buf = getcallbuf (rpcm);
  XDR *xdr = xdrmem_create (....);
  switch(op) {
    /* Decode and set fh based on op */
 }
  return fh;
}
```

Listing 2: Extraction operator code for extracting FH from the request message

FWL supports all data types defined by the native language as well as those defined in the templates. Variables can be declared by assignment and their type is determined automatically during compilation. If the type cannot be determined, for example, when using a variable without prior assignment, an error is generated.

5.2 Attribute Extraction

FileWall policies transform attributes. Therefore, attribute extraction is the basic building block of all policies. Attributes are present in packets as a stream encoded in a host-independent data representation format. All file system messages follow this stream format, and the protocol templates specify the encode and decode routines.

FileWall provides an extraction operator "@" that incrementally decodes the packets and presents the policy writer with a reference to the memory containing the decoded attribute. The type of the decoded attribute is defined by the attribute being requested as defined in the protocol templates.

The arguments to the @ operator are the message and the names corresponding to the attribute name, for example, the FWL statement to extract the file handle from a request message is `FH = FHREQ@MSG`. Similarly to assign a file handle to a request we can use `FHREQ@MSG = FH`. FWL translates the former to a call to the function `extract_FHREQ` and the latter to `assign_FHREQ`.

The FileWall templates define the extraction and assignment functions with names of the form `extract_X` and `assign_X` where X can be used as the left hand side of the extraction operator.

A subset of the C code that implements `extract_FHREQ` is shown in Listing 2. In the function, the operation type is extracted from the request message. If the operation specifies a file handle, the protocol translation routine for the operation is called to extract the decoded request data structure. Finally, the file handle is returned. If the message does not specify the file handle a NULL is returned. A corresponding assignment function sets the value of the attribute.

5.3 Associative Arrays

Associative arrays are declared in the configuration section with the appropriate type information. By default, the String type is used as the key, and the opaque type is used for the data. For associative arrays with more than one dimension, the indices are concatenated to identify the index of the element. The first statement that assigns an element to the array is used as the type of the array elements.

By default, associative arrays are stored in the dynamic context of a policy and are not persistent. An in-memory hash table is

used to implement these arrays. However, if a special operator, #, is used in the configuration section when declaring an associative array, it is stored persistently on stable storage. FileWall uses a database with the same name as the array. The indices are used as keys and the elements as the data of records stored in the database. Reading an element in persistent associative arrays translates to a database lookup and an assignment to a database update.

6. FileWall POLICIES

In this section, we describe FWL implementations of four policies introduced in Section 2. The following policy specifications are the actual implementation of policies for NFS in our system. Each specification translates to more than 1000 lines of C++ code excluding comments, packet capture and injection code, and the NFS templates. Without FileWall, the administrator would have to understand the NFS implementation, modify, and debug the OS specific code for servers and clients.

6.1 Statistics Monitoring

In the following, we describe a Statistics Monitoring policy implemented with FileWall. This policy illustrates attribute extraction and the access context usage in a FileWall policy specified in FWL.

Access context: This policy only requires non-persistent state storage in the access context. The state generated by the statistics policy is counters associated with each client-server pair, and is stored in three associative arrays: the `reqstats`, `rspstats`, and `bytecount`.

A record in the `reqstats` or `rspstats` arrays represents the count of an operation in the request and response streams respectively. The `bytecount` array records the number of bytes of data written or read by the client. The following FWL code initializes these arrays and the policy itself. The index types are defined in FileWall and NFS templates.

```
@CONFIG::stats {
  int reqstats[ IPFlowID][ nfs3_ops] ;
  int rspstats[ IPFlowID][ nfsstat3] ;
  int bytecount[ IPFlowID][ nfs3_ops] ;
}
```

Request Handler: For each request, the statistics policy extracts the flow identifier and operation number from the message. It updates the operation count stored in the `reqstats` array. Finally, in the case of a WRITE operation, the policy increases the bytecount by the size of the buffer sent to the server in the request.

```
@REQ {
  op = op@$MSG;
  flowid = flowid@$MSG;
  reqstats[ flowid][ op] ++;
  if(op == NFSPROC3_WRITE) {
    bytecount[ flowid][ NFSPROC3_WRITE]  += count@$MSG;
  }
  forward($MSG);
}
```

Response Handler: For each response, the statistics policy extracts the flow identifier, operation number, and the status returned by the server. Note that while the operation number is not available in the response message, the FileWall attribute extraction uses the matching request to find the operation identifier. The update to the `rspstats` array is analogous to the request handler. Finally, in the case of a READ operation, the statistics policy increases the bytecount by the size of the buffer sent to the server in the request.

```
@RSP {
  op = op@$MSG;
  flowid = flowid@$MSG;
  rspstats[ flowid][ op] ++;
  if(op == NFSPROC3_READ) {
    bytecount[ flowid][ NFSPROC3_READ]  += count@$MSG;
  }
  forward($MSG);
}
```

6.2 Temporal Access Control

In the following, we describe a temporal access control policy implemented with FileWall. This policy prevents all accesses to the file system during a specified time window. This policy illustrates an extended ACL attribute, time, being added to the default `rwx` access control. This attribute is maintained only by FileWall and is not visible either to the client or the server.

Access context: This policy requires current time from the environment and the time window in the access context. The time window is specified during configuration as start_time and end_time.

```
@CONFIG::TemporalACL {
  start_time = 1200;
  end_time = 1300;
  extern denyrsp;
  extern random;
}
```

Handlers: For each request, this policy extracts the current time from the environment. Between the start_time and end_time (specified in the 24 hour clock format) the policy generates a deny message in response to the client request using the `denyrsp` external function. The response handler simply forwards the message. FileWall classifies the forwarded response, identifies the corresponding request message ($MSG), and discards the original request message. The FWL code for this policy is shown below.

```
@REQ {
  if(($curtime > start_time) &&
      ($curtime < stop_time)) {
    rsp = denyrsp($MSG);
    forward(rsp);
    return;
  }
  else
    forward($MSG);
}
@RSP {
  forward($MSG);
}
```

6.3 File Handle Security

In the following, we describe a file handle security policy. This policy generates per-client virtual file handles and stores a mapping between the virtual and real (server generated) file handles. Client requests that contain file handles not generated by FileWall are rejected without sending them to the server. For all other messages, FileWall replaces the virtual file handle with the server file handle.

Access context: This policy uses both static and dynamic access context. For requests and responses, the incoming file handle is extracted from the message. The policy also maintains two maps as dynamic access context (forward and reverse file handle to virtual file handle maps).

```
@CONFIG::FHSecurity {
  ...
  nfs_fh3 #fwdmap[ nfs_fh3] ;
  nfs_fh3 #revmap[ nfs_fh3] ;
  extern denymsg;
}
```

Request Handler: For each request, this policy extracts the virtual file handle (`vfh`), if it exists, from the message. The policy performs a lookup in the dynamic access context (`fwdmap`), using

the vfh, to determine the real server file handle (fh). Once found, the request is transformed to replace the vfh with the fh. For any vfh to fh lookup performed, if a fh is not found, a deny response is generated and the original message is dropped.

```
@REQ {
  ...
  vfh = fhreq@$MSG;
  fh = fwdmap[ vfh] ;
  if(FH == NULL) {
    rsp = denymsg($MSG);
    forward(rsp);
    return;
  }
  fhreq@$MSG = fh;
  forward($MSG);
}
```

Response Handler: For each response, this policy extracts the server file handle (fh), if it exists, from the static access context (response message). If the fh is not passed in by the response (e.g., ACCESS responses), the policy exits, passing the message on to the next policy in the chain. Otherwise, the policy performs a lookup in the dynamic access context (revmap), using the fh, to determine the virtual file handle (vfh). If the vfh does not already exist, one is generated by the random externally defined function and the new mapping inserted into fwdmap and revmap. Once a vfh is obtained, the response is transformed to replace the fh with the vfh.

```
@RSP {
  fh = fhrsp@$MSG;
  vfh = revmap[ fh] ;
  if(vfh == NULL) {
    vfh = random;
    revmap[ fh]  = vfh;
    fwdmap[ vfh]  = fh;
  }

  fhrsp@$MSG = vfh;
  ...
}
```

For the READDIRPLUS operation, the file handles returned by the server in the entry list must also be transformed. The for loop in the following iterates through the list of entry objects transforming fhs to vfhs in each entry.

```
@RSP {
  ...
  entrylist = entries@MSG;
  for(i = 0; i < entrylist.nelem; i++)
  {
    fh = fh@entrylist[ i] ;
    /* Generate virtual file handles */
    fh@entry = vfh;
  }
  forward($MSG);
}
```

6.4 Client Transparent Failover

In the following, we describe a replication policy, which generates a copy of client requests for each replica. Virtual file handles are used to maintain a one-to-many mapping across replicas. We assume all servers start with identical state and the failure mode is fail-stop. That is, once a server fails, it must be brought back to a consistent state external to the normal file system operation. Server responses are forwarded to the clients only after a response is received from each of the replicas.

Access context: This policy uses a list of replicated file servers specified in the policy configuration. Each server is assigned an identifier and a liveness bitmap is maintained in the environment. The failover policy generates virtual file handles for each client-server pair similar to the file handle security using forward and reverse maps.

```
@CONFIG::Failover {
  struct server {
    int id;
    nfs_fh3 #fwdmap[ nfs_fh3] ;
    nfs_fh3 #revmap[ nfs_fh3] ;
  } ;
  server servermaps[ IPAddr] ;
  int pendingreqs[ XID] ;
  replicas = { s0, s1} ;
  extern copymsg;
}
```

Request Handler: This policy generates a new message for each live server in the server list. The new message is a copy of the incoming request message. The file handle is replaced similar to the file handle security policy. However, in this policy, each server has its own vfh to fh map and the message is forwarded to all live servers.

```
@REQ {
  vfh = fhreq@$MSG;
  XID = XID@$MSG;
  msglist = [] ;
  foreach s in replicas {
    server = servermaps[ s] ;
    fh = server.#fwdmap[ vfh] ;
    reqmsg = copymsg($MSG);
    fhreq@reqmsg = fh;
    dstaddr@reqmsg = s;
    msglist.append(reqmsg);
  }
  ...
  bmap = 0;
  foreach m in msglist {
    forward(m);
    saddr = dstaddr@m;
    srv = servermaps[ saddr] ;
    bmap |= (1 << srv.id);
  }
  pendingreqs[ XID]  = bmap;

  msglist.clear();
}
```

Response Handler: The response handler performs two main functions. First, it suppresses multiple responses for the same request. FileWall forwards the response to the client after all responses are received from the servers. However, this may easily be modified to only wait for the first response or until a majority of responses have been received. Second, it populates the per-server vfh to fh maps on receiving the file handle information, e.g., in LOOKUP and READDIRPLUS responses.

```
@RSP {
  saddr = srcaddr@$MSG;
  XID = XID@$MSG;
  bmap = pendingreqs[ XID] ;
  srv = servers[ saddr] ;
  bmap &= ~(1 << srv.id);
  if(bmap == 0) {
    /* Replace fh with vfh */
    ...
    forward($MSG);
  }
  else {
    discard($MSG);
  }
}
```

7. IMPLEMENTATION

We implemented FileWall at the user-level as an element of the Click modular router [16]. Implementing FileWall at the user-level

allows for easy debugging and platform independence. FileWall relies on Click only for capturing packets from the network and reinjecting packets back into the network. In principle, FileWall can work over any packet capture and injection framework. We chose Click for simplicity and portability – our system works unmodified on two different operating systems: Linux and FreeBSD.

We implemented the access context using an open-source database – BerkeleyDB [12]. This database shares the process address space allowing direct function calls to be made to the DB. We use the default DB configuration where the data is stored as B-Trees. Separate databases are created, for each policy in the system, to maintain private access context.

Policies are loadable shared objects that are dynamically loaded into the FileWall system during operation. At initialization, FileWall defines two default policies *SEND* and *RECV* policies and all messages are simply passed through to the destination without transformation. We provide handlers that load policies during execution and the associated administrative tools that interface with the Click handler invocation framework.

The FileWall templates define the skeleton of the classes and the initialization code required to construct a Click element class. The FWL compiler is implemented using treecc [28], which converts input files in the treecc syntax into source code that permits creating and walking abstract syntax trees. We use treecc along with standard compiler generation tools, lex and yacc, to generate C++ code from the FWL code.

8. EVALUATION

In this section, we present an evaluation of our system. Our goal in this evaluation is to measure FileWall *overheads* and to characterize FileWall *behavior*, under varying network conditions and workloads.

We first measure the interposition overheads and study the effect of FileWall placement (at the server, client, and interposed on the network path) on the client perceived performance through microbenchmarks. Next, we characterize how FileWall behaves as a number of policies are added to it.

After characterizing our system through microbenchmarks, we study its behavior under varying network characteristics. Finally, we compare the performance of FileWall against default NFS for the same setup using two real-world workloads – a software compilation, and Fstress [1], a self-scaling benchmark which generates workloads similar to SpecSFS97.

8.1 Setup

In our experimental setup, all systems are Dell Poweredge 2600 SMP systems with two 2.4GHz, Intel Xeon II CPUs with 2GB of RAM and 36GB 15K RPM SCSI drives. All systems run Fedora Core 3 distribution with a Linux-2.6.16 kernel. We use the Click 1.5.0 distribution available online. FileWall is configured to interpose on all NFS requests and responses, and stores the access context using the BerkeleyDB-4 distribution, available online. All systems are connected using a Gigabit Ethernet switch, and the round-trip time between any two hosts is $30\mu s$.

8.2 Microbenchmarks

To study the behavior of the file system, with and without FileWall, we developed our own microbenchmark. This benchmark is an RPC client and issues NFS requests *without* relying on the client file system interface. Using this benchmark eliminates the noise due to client buffer cache and other file system optimizations, and allows fine-grained measurements. The benchmark measures the CPU cycles between a call and the corresponding response.

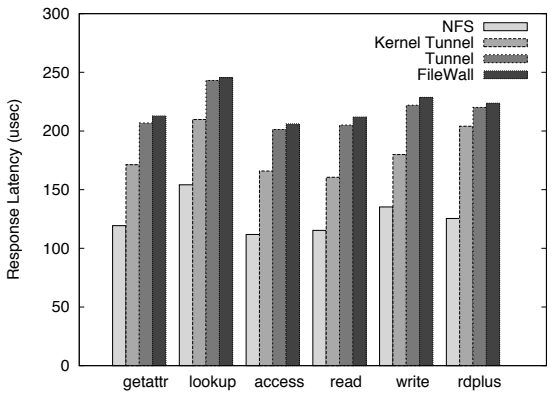

Figure 7: Interposition Overheads.

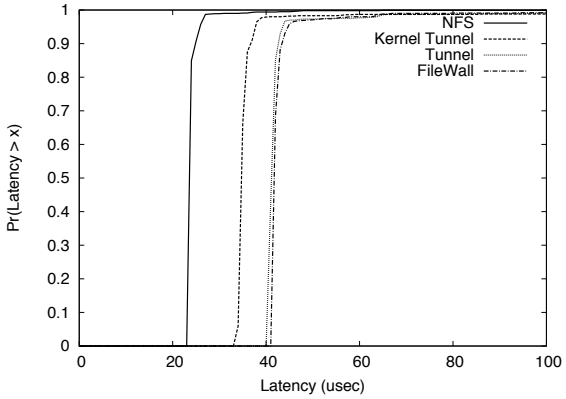

Figure 8: CDF for individual call times for READDIRPLUS.

Interposition Overheads: FileWall introduces a network element in the client-server path. The overheads imposed by FileWall are two-fold. First, the interposition overhead due to all packets passing through FileWall, and second, the overheads of the policies and message transformation.

We implemented FileWall at the user level, which imposes additional overheads of capturing and injecting packets. The user level overheads are higher than an in-kernel implementation due to additional copying and scheduling delays. However, a user level implementation significantly reduces the development time, is portable, and as we show in the following sections, the overheads are reasonable for the workloads we studied.

Figure 7 shows the interposition overheads imposed by FileWall-compared to the default NFS implementation for different NFS operations. For clarity, we show the most common operations, as reported by various file system workload studies [7]. In the figure, each group of bars has 4 members, base NFS, tunnel, kernel tunnel, and FileWall. The height of each bar shows the average response latency for 1000 instances of the call. For this experiment, we use the file handle security policy for FileWall. We observed similar results for other policies as well as for the NFS operations not shown in the figure.

Tunnel implements a pass-through network tunnel – all packets from the client are captured and forwarded to the server without any processing. The *kernel tunnel* is an in-kernel implementation

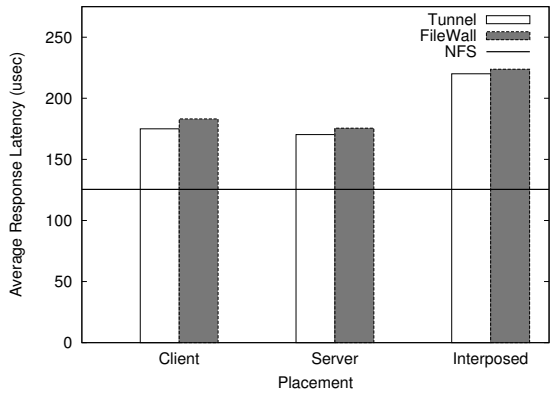

Figure 9: Overheads of placing FileWall at client, server, and interposed

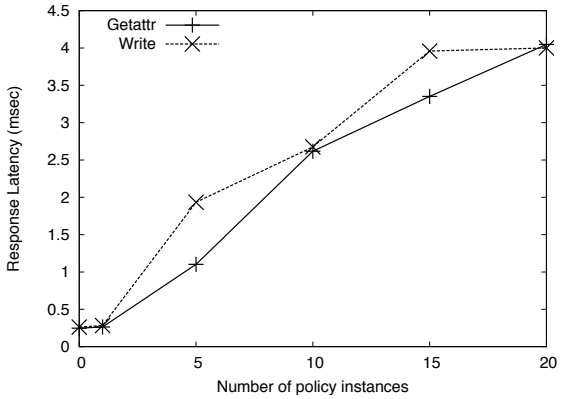

Figure 10: FileWall scalability with increasing number of policies.

of the tunnel. These represent the base interposition overheads. The difference between these and the FileWall time represent the overheads of policy execution.

We observe that the base interposition overheads are within $100\mu s$ for the user level tunnel implementation. FileWall imposes minimal overhead compared to the interposition overheads. Finally, as expected, the kernel-tunnel is more efficient than the user level implementation, but the difference is less than $5\mu s$. Figure 8 shows the cumulative distribution function (CDF) for all LOOKUP calls to illustrate that these overheads do not vary significantly across measurements and are fixed. The additional variation for the interposed calls is due to the scheduling delays at the FileWall.

FileWall Placement: FileWall can be instantiated at the server, at the client, or on a separate machine in the network path. We measure the overheads of placing the user level tunnel and FileWall at each of the above using our microbenchmark.

Figure 9 shows the results for our microbenchmark for the READDIRPLUS call. We use the file handle security policy for FileWall. The figure shows the average time between the call and its response at the client when the tunnel and FileWall are instantiated at the client, server, and on a separate machine on the network.

We observe that *(i)* the overheads of FileWall compared to the tunnel are low, therefore, FileWall policies do not add additional overheads, and *(ii)* FileWall instantiation on a separate node (our model) performs well. The average time for this case is within $50\mu s$ of FileWall instantiated on the server or the client. This overhead is a property of our topology, where all machines, the client, server, and FileWall are connected to the same switch and the one-way delays are less than $20\mu s$. In a more realistic scenario, FileWall-server delays would be much less than the client-FileWall delays and the overheads would be hidden.

Scalability: The number of FileWall policies vary across deployment. Therefore, to understand the client perceived performance as the number of policies increases is difficult. We define a policy that captures the tasks common across a wide range of policies and vary the number of instances of such policies to study FileWall behavior.

Each policy performs the following tasks: On a request message, the policy stores a fixed number (20) keys, each of size 50 bytes in the access context (DB insert). On a response, it looks up all the keys inserted by the corresponding request and deletes them (DB lookup and delete). Since DB access is the most CPU intensive task performed by a FileWall policy, our results capture the expected behavior. However, these results may vary with different policies.

Figure 10 shows the average response latency as the number of policies is varied. The two curves are for GETATTR and WRITE requests, which illustrate the FileWall performance for metadata and data operations respectively. We observe that FileWall overheads increase linearly in the worst case and even with 20 policies, the response time is within 5 ms.

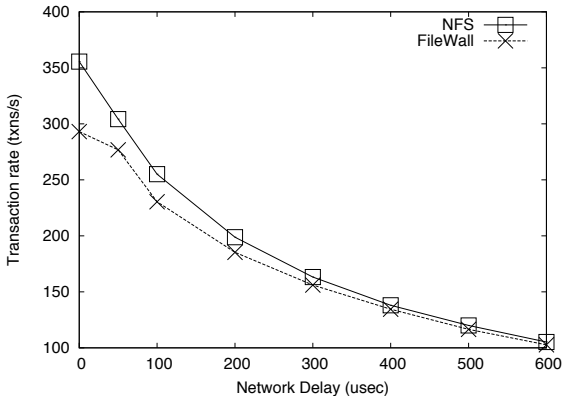

Figure 11: Postmark with varying network delay

8.3 Delay

Postmark [13] is a synthetic benchmark that measures file system performance with a workload composed of many short-lived, relatively small files. Postmark workloads are characterized by a mix of metadata intensive operations. The benchmark begins by creating a pool of files, performs a sequence of transactions, and concludes by deleting all of the files created. Each transaction consists of two operations: a randomly chosen CREATE or DELETE, paired with a randomly chosen READ or WRITE.

In our experiments, we use 8KB block sizes for read and write operations for Postmark. The initial file set consists of 5,000 files with sizes distributed randomly between 1KB and 16KB. For each run of the benchmark, we perform 20,000 transactions and report the transaction rate. Figure 11 shows the Postmark results for varying network delay between the client and the server for base NFS and for FileWall with the file handle security policy. Other policies demonstrate similar behavior and are omitted for clarity. For File-

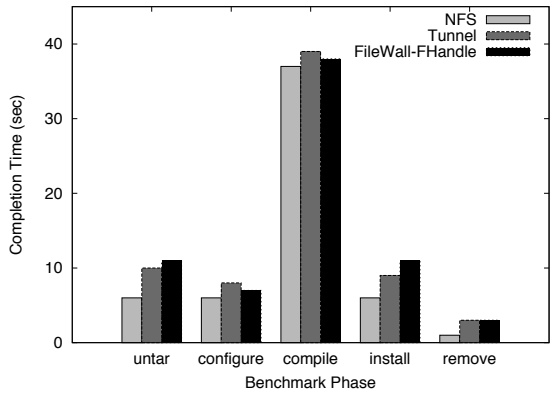

Figure 12: Results of emacs compilation benchmark.

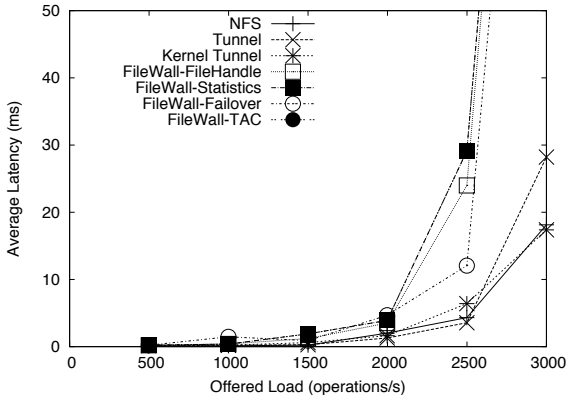

Figure 13: Fstress performance with 2.4GHz CPU.

Wall delays are introduced on the client-FileWall link, while the FileWall-server link has the default delay.

We observe that for low delays ($< 100\mu s$), FileWall and the base NFS differ significantly in the supported transaction rate. However, for networks with larger delays, the difference between FileWall and NFS performance is minimal. FileWall introduces delays of the order of 10s of μs, therefore, for comparable network delays, the performance of latency sensitive operations is greatly affected. However, these delays are hidden when the network delays dominate the overall delay between the client and the server.

For typical deployments, file servers are physically separated from clients and delays of up to $300\mu s$ are common. Therefore, in realistic scenarios, we believe that FileWall will not affect performance and will be transparent to the clients.

8.4 Real Workloads

Emacs Compilation: In the following, we compare the performance of FileWall with the file handle security policy with default NFS and a pass-through tunnel for a multi-stage software build similar to a modified Andrew Benchmark [10]. We measure the time taken to *untar*, *configure*, *compile*, *install*, and *remove* an Emacs 20.7 distribution.

The Emacs distribution size is 76MB, which reduces to 20MB when tarred and gzipped. It contains 43 directories and 76,400 files. The compilation performs a large number of read, write, lookup, create, and remove operations. At the end of the compilation, the total number of files in the directory is 101,644 and the size is 95MB. We performed one run of the benchmark to load the compiler binaries and associated libraries that are external to the file system under test. We discarded this result and performed ten further runs. Between each run of the benchmark, we unmounted the file system and mounted it to start with a cold cache on the file system under test at the client.

Figure 12 shows the time taken for each phase of the Emacs compilation benchmark from left to right. The bars in each group are NFS, Pass-through tunnel, and FileWall with the file handle security policy. Experiments with other policies described in the paper yielded similar results and are omitted for clarity.

We observe that both FileWall and Tunnel take from $10 - 40\%$ additional time for untar, configure, install, and remove phases. For the most expensive compile phase of the benchmark, FileWall imposes less than 12% overhead. Overall, the performance is within 15% of the base NFS. In all cases, FileWall imposes minimal overheads compared to the pass-through tunnel.

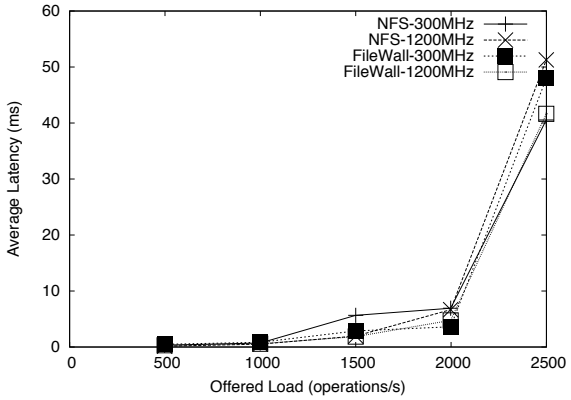

Figure 14: Fstress performance with different NFS server CPU speeds.

These results are for a network with no additional delay. As shown in the microbenchmarks, the reported results are the most pessimistic for FileWall. For networks with larger delays, the performance of all three – NFS, tunnel, and FileWall is identical.

Fstress: Fstress [1] is a synthetic, self-scaling benchmark that measures server scalability. We use the canned SpecSFS97 like workload distributed with Fstress, which performs random read-write access with file sizes varying from 1KB–1MB. The workload is characterized by a large number of directories with thousands of files in each directory. The size of the file set accessed by Fstress is adjusted to reflect the offered load on the system. The operation distribution is identical to that defined by SPECsfs, which in turn is based on a survey of file set distributions and workloads on one thousand NFS servers.

Fstress increases the offered load by issuing NFS requests according to the workload mix. The metric used is the latency of these operations. For an overloaded server, the client observed latencies grow rapidly, whereas for an underloaded server, these latencies are small. For our experiments, we use three load generators (clients) which have configurations described in Section 8.1. The client generators are connected to the same Ethernet switch as the servers as well as FileWall.

Figure 13 shows the results from the Fstress benchmark. We observe that FileWall latencies are comparable to both the default

NFS and the network tunnels (user and kernel) up to an offered load of 2,500 requests/s. The systems diverge rapidly beyond this point. However, in all cases, the NFS server is overloaded at around 3,000 requests/s. The sharp increase in observed latency with FileWall is due to the CPU at FileWall being overloaded. At overload, the small processing overheads imposed by FileWall become significant and the performance drops. However, the overheads are small (around 15%) and the enhanced functionality provided in exchange by FileWall make them acceptable.

FileWall does not necessarily saturate before the NFS server. The impact of FileWall under heavy load is due to the relative performance between the FileWall machine and the NFS server machine. This is illustrated in Figure 14. For this experiment we run Fstress, while varying the CPU speed of the NFS server. We use the Intel Speedstep voltage scaling to reduce the CPU speed to 300MHz and 1200MHz, while maintaining all other system parameters constant. The curves in the Figure represent the base NFS and FileWall performance for the different NFS server CPU speeds. In all cases, we observe that the performance is similar with and without FileWall, and the NFS server saturates beyond 2000 requests/s. We conclude that given sufficient resources relative to the NFS server, FileWall does not impose significant overheads even under heavy workloads.

9. RELATED WORK

FileWall is related to a variety of work in the areas of *(i)* Distributed and Extensible File Systems, *(ii)* Extensible Policies, and *(iii)* Composable Network Processing. The following section is a survey of the work related to FileWall in these three areas.

Distributed and Extensible File Systems: Several proposals for extending local file systems by interposing on the request path have been proposed [30, 29]. These systems interpose between the file systems and transform the vnode operations to enhance functionality. Such systems have been used to trace file system calls [4], build versioning file systems [21], and even virus protection [19]. Unfortunately, these systems operate at the vnode interface. Therefore, to implement network file system policies, server or client systems must be modified. In contrast, FileWall implements policies by operating on the file system messages external to both file servers and clients. It also provides an interface for administrators to implement policies on a single system, thereby enabling instantaneous global policy enforcement.

Network storage systems have been extended for functional decomposition [2], enhanced security [17], saving bandwidth [22, 3], repair and rollback [31], extended access control [9], and using interposing proxies. FileWall focuses on implementing file system policies. Unlike other systems, where the policies are client-centric, we target administrator defined system-wide policies. FileWall provides a policy enforcement framework where each of the above can be specified easily, external to both client and server systems.

FiST [30] is a stackable template language, which shares a similar structure with FWL. FiST templates are used in conjunction with a Base file system to extend the operating system vnode interface. Similar to FWL, FiST provides environment variables and supports extended attributes for file system. However, unlike FiST that operates on local file system calls and a stackable vnode interface, FWL operates in context of network file system messages and relies on standardized data representation for attribute extraction.

Extensible Policies: SPIN [25] and VINO [23] are extensible operating systems that allow applications to make policy decisions and modify operating system interfaces by safely downloading extensions into the kernel. SPIN ensures safety using a type-safe language Modula-3, while VINO implements mechanisms for fault isolation. Unlike the extensible OS, FileWall does not modify OS interfaces, targets network file system policies, and implements the policies external to the client and server systems.

Exokernel [8] exports fine-grained hardware services, for example, TLB management, directly to applications. Exokernel provides no abstractions beyond those minimally provided by the hardware and allows libraries to implement OS policies. FileWall policies are similar in spirit to the Exokernel stable storage system XN. To implement file systems, XN allows users to define on disk data structures and methods to implement them with libraries (libFSes). FileWall uses the minimal abstractions provided by the message streams to implement network file system policies and allows administrators to define policies.

Infokernel [5] provides operating system mechanisms to modify existing policies and implement new policies. In contrast, FileWall operates in a different domain by using the information contained in file system messages and the *access context* to implement new file system policies.

Law Governed Interaction(LGI) [18] proposes a distributed policy definition and enforcement framework for heterogeneous distributed systems. In LGI, policies are defined for control and coordination of resources, which range from web-services to file systems. The actions of principals (users or servers) are controlled by an external LGI monitor. Although, FileWall provides equivalent mechanisms to specify and enforce access control policies, performance requirements make it impossible to support external policy enforcement in the critical path. In addition, FileWall supports the specification of a wide range of policies, beyond access control.

Composable Network Processing: FileWall is inspired by packet filters [6], network address translators, and firewalls, which implement network access policies by interposing on network traffic and mapping private addresses to public Internet addresses.

The x-kernel [11] is a framework for implementing and composing network protocols. An x-kernel configuration is a graph of processing nodes, and packets are passed between nodes through function calls. Unlike FileWall, where policies are composed as chains and file system attributes are transformed during processing, the x-kernel nodes are arranged in an acyclic graph and the inter-node communication is more complex than a simple function call.

The Scout Operating System [20] is an OS architecture where *paths* defined by a sequence of processing nodes traversed by a network packet are made explicit. Packets are classified into the correct path as early as possible, so that packets are treated differently as soon as they arrive on the host. In addition, Scout paths support different kinds of inter-node communication beyond a simple packet flow. In contrast, FileWall handles only file system messages and all messages flow through the same policy chain.

Click [16] implements a modular router architecture, where different components of router processing are implemented independently and composed in a flow based context through virtual function calls. Click supports explicit queues and asynchronous *pull* processing, which enable complex router implementations. In contrast to Click, which processes packets, FileWall processing is in context of file system messages, which may be composed of multiple packets. FileWall also supports persistent state using a database, which is not required for a router.

10. CONCLUSIONS

We have presented the architecture, design, and implementation of FileWall, a proxy, which implements network file system policies through message transformation. FileWall provides a plat-

form for administrators to define enterprise-wide policies at a single point in the network, without modifying the clients or the servers.

We have also designed FWL, a high level policy specification language. Administrators can easily specify policies using FWL, without being encumbered by OS specific details of protocol implementation.

We have demonstrated the versatility and ease of use of FileWall through the implementation of four real-world policies. These policies implement statistics monitoring, access control, security, and client transparent failover beyond those supported by existing file systems. Policies implemented with FileWall are portable and do not rely on specific operating system implementations of the file system protocols.

We have implemented a FileWall prototype at the user level using the Click modular router. Our evaluation shows that even with a user level implementation, the interposition overheads are low. In the worst case, when network delays are as low as $30\mu s$, FileWall causes a 15% performance degradation for latency sensitive workloads. We also show that FileWall scales linearly with the number of policies. Finally, given sufficient resources relative to an NFS server, we show that FileWall imposes minimal overheads even under heavy workloads.

11. REFERENCES

[1] D. Anderson and J. Chase. Fstress: A flexible network file service benchmark, 2001.

[2] D. C. Anderson, J. S. Chase, and A. Vahdat. Interposed request routing for scalable network storage. *ACM Trans. Comput. Syst.*, 20(1):25–48, 2002.

[3] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling File Servers via Cooperative Caching. In *Proc. of NSDI'05*, Boston, MA, May 2005.

[4] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proc. of FAST*, San Francisco, CA, March/April 2004. USENIX Association.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proc. of SOSP*, October 2003.

[6] H. Bos, W. de Bruijn, M.-L. Cristea, T. Nguyen, and G. Portokalidis. Ffpf: Fairly fast packet filters. In *Proc. of OSDI*, 2004.

[7] D. Ellard, J. Ledlie, P. Malkani, and M. I. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proc. FAST*, San Francisco, CA, 2003.

[8] D. Engler, M. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of SOSP'95*, December 1995.

[9] Z. He, T. Phan, and T. D. Nguyen. Enforcing enterprise-wide policies over standard client-server interactions. In *Proc. of SRDS*, oct 2005.

[10] J. H. Howard et al. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[11] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. In *IEEE Trans. on Soft. Eng.*, 1991.

[12] S. S. Inc. Berkeley db. `http://dev.sleepycat.com/`.

[13] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance Inc., October 1997.

[14] CERT Common Vulnerabilities and Exposures - CVE-2006-3468. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3468`.

[15] CERT Common Vulnerabilities and Exposures - CVE-2005-3623. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3623`.

[16] E. Kohler et al. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[17] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. of SOSP*, dec 1999.

[18] N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol.*, 9(3), 2000.

[19] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An On-Access Anti-Virus File System. In *Proc. of Security 2004*, San Diego, CA, August 2004. USENIX Association.

[20] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system (abstract). In *Operating Systems Design and Implementation*, page 200, 1994.

[21] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proc. of FAST*, San Francisco, CA, March/April 2004.

[22] A. Muthitacharoen, B. Chen, and D. Maziéres. A low-bandwidth network file system. In *Proc. of SOSP*, New York, NY, USA, 2001.

[23] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. 2nd Symp. on Operating Systems Design and Implementation (OSDI)*, Oct. 1996.

[24] M. Sheldon, D. Gifford, P. Jouvelot, and J. O. Jr. Semantic file systems. In *Proc. of SOSP*, Pacific Grove, CA, 1991.

[25] E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Dec. 1995.

[26] C. Soules and G. Ganger. Connections: using context to enhance file search. In *Proc. of SOSP*, Brighton, UK, 2005.

[27] A. Traeger, A. Rai, C. P. Wright, and E. Zadok. NFS File Handle Security. Technical Report FSL-04-03, Computer Science Department, Stony Brook University, May 2004.

[28] R. Weatherley. An aspect oriented approach to writing compilers. `http://www.southern-storm.com.au/treecc.html`.

[29] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM TOS*, 2(1), March 2006. To appear.

[30] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of USENIX*, San Diego, CA, June 2000. USENIX Association.

[31] N. Zhu, D. Ellard, and T.-C. Chieuh. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *Proc. of FAST*, San Francisco, CA, December 2005.