5-2011

SCALABLE CAPABILITY-BASED AUTHORIZATION FOR HIGH-PERFORMANCE PARALLEL FILE SYSTEMS

Nicholas Mills
Clemson University, nlmills@g.clemson.edu

SCALABLE CAPABILITY-BASED AUTHORIZATION FOR HIGH-PERFORMANCE PARALLEL FILE SYSTEMS

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Nicholas L. Mills
May 2011

Accepted by:
Dr. Walter B. Ligon III, Committee Chair
Dr. Richard R. Brooks
Dr. Adam W. Hoover

Abstract

As the size and scale of supercomputers continues to increase at an exponential rate the number of users on a given supercomputer will only grow larger. A larger number of users on a supercomputer places a greater importance on the strength of information security. Nowhere is this requirement for security more apparent than the file system, as users expect their data to be protected from accidental or deliberate modification.

In spite of the ever-increasing demand for more secure file system access the majority of parallel file systems do not implement a robust security protocol for fear it will negatively impact the performance and scalability of the file system. We provide a capability-based security protocol for use in high-performance parallel file systems that is capable of meeting the performance and scalability requirements of current and future supercomputers. We develop a reference implementation for the Parallel Virtual File System [3] and show its performance characteristics using several microbenchmarks. Our test results show that capability-based security is capable of protecting access to parallel file system objects, in some cases with little overhead.

Acknowledgments

First, I would like to thank my advisor, Dr. Ligon, for his support and guidance during the development of this thesis. Similarly, I would like to thank Dr. Brooks and Dr. Hoover for serving on my committee and providing me with valuable feedback. Much of the design and implementation of the proof-of-concept code was developed jointly with David Bonnie, and I would like to thank him for this effort. Finally, I would also like to thank Mike Marshall and Michael Moore for their assistance in creating and analyzing test data.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The performance of large parallel compute clusters continues to increase every year thanks in part to a steady increase in the performance of the commodity processors used in these systems. In addition, the degree of parallelism available to individual compute nodes is also increasing as the nodes contain a proliferating number of processing cores. For example, the top three systems on the TOP500 list have hundreds of thousands of cores and a peak performance of well over one petaflops [14]. GPGPU technology, a relative newcomer to the high-performance computing scene, is also playing a larger role with the ready availability of technologies such as CUDA and OpenCL [15, 11].

Such large increases in the processing power of supercomputers requires a complimentary increase in the performance of I/O systems responsible for keeping the compute cores fed with data. Scientific applications that run on supercomputers make frequent demands of the file system and are known for their bursty access patterns and large request sizes [23]. Other domains, such as the MapReduce search framework, also require fast access to large amounts of data [6]. Parallel file systems were developed to meet these demands.

1.1 Parallel File Systems

Parallel file systems developed from the more general case of distributed file systems. One example of an early distributed file system that is still in use today is the Sun Network File System (NFS) [18]. NFS has the network architecture shown in Figure 1.1. In NFS the client nodes mount

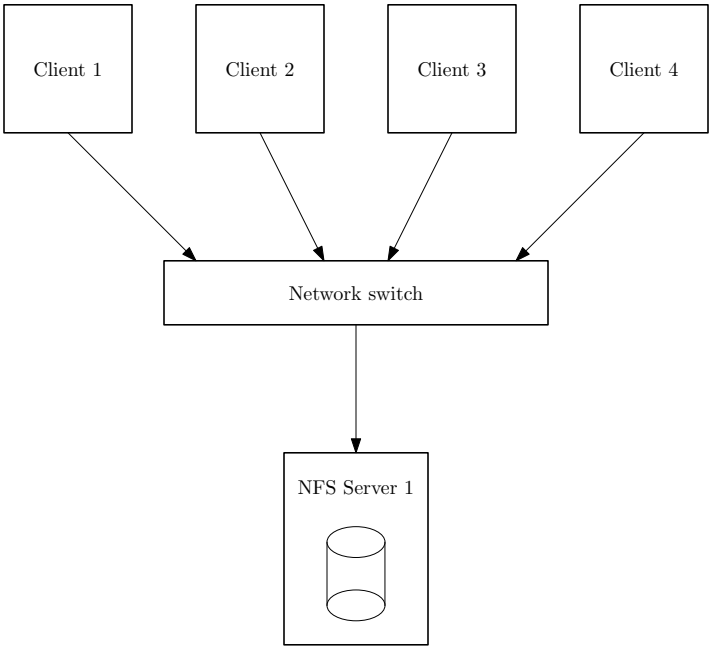

Figure 1.1: Network containing an NFS server

a file system that is located on a single server node. This server node handles all metadata and I/O requests from multiple clients. The resulting system does not scale well to meet the demands of high-performance computing applications because a centralized file server is feasible only when both the number of clients and the I/O workload are relatively small.

In contrast, parallel file systems were designed to handle large I/O requests from large numbers of clients. Parallel file systems have the network architecture shown in Figure 1.2. While the parallel file system servers present a single file system image to clients, actual file data is striped across the available servers. This distribution of file data among the various servers is what allows file operations to proceed in parallel.

1.2 The Case for More Robust Security

Early parallel file system implementations were concerned first and foremost with performance. Closing the gap between compute power and I/O performance was critical to the success of previous supercomputing systems. Robust security was considered a secondary concern because of its perceived negative effects on the performance and scalability of the file system. The existing norm of secure compute clusters shared by a small, trusted group of users somewhat obviated the

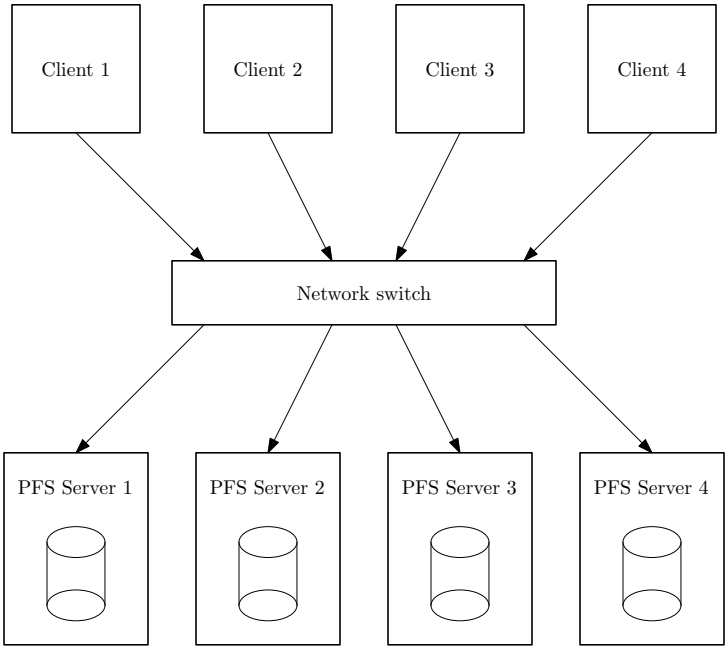

Figure 1.2: Network containing parallel file system servers

need for robust security at the file system level.

The size and cost of modern compute clusters contrasts sharply with earlier supercomputers. The fastest computer on the first TOP500 list in 1993 was the CM-5 at 1024 cores, whereas the modern Jaguar cluster at Oak Ridge National Laboratory consists of 224,162 cores and cost more than $100 million [14, 4]. Clearly it is not feasible for such large and expensive systems to be used by a single group. Instead, a single large compute cluster may be used simultaneously by multiple organizations and across geographic boundaries.

The need for more robust security in a parallel file system increases along with the size and diversity of the user base. Users expect that their data are protected from unwanted modifications, be they deliberate or accidental. Thus a greater emphasis is placed on robust security for parallel file systems than in the past. The challenge lies with designing a security mechanism that can protect the integrity of data without impacting the performance and scalability of the file system.

1.3 Capabilities As a Solution to the Security Problem

The greatest practical challenge to the development of a robust security mechanism for high-performance parallel file systems is a consequence of the basic architecture of those file sys-

tems. Parallel file systems achieve performance gains by splitting data across multiple I/O nodes. These I/O nodes are logically and often physically separate from the metadata nodes that store the information necessary to control access to data. While it is possible for I/O nodes to communicate with metadata nodes over the network, doing so is discouraged because it limits the scalability of the file system. Thus the I/O nodes that perform the bulk of file system operations are unable to perform many security checks because they have no knowledge of the authorization information stored on the metadata servers.

Capability-based security is a convenient and powerful mechanism for transferring authorization information from the metadata servers to the I/O servers through the client. Before clients can access a resource on a file system using capability-based security they must first acquire a capability for that resource. The general form for a capability can be represented with the pair

$$(\text{object}, \text{rights})$$

where *object* is a reference to the resource being accessed and *rights* is a set of operations the owner of the capability is allowed to exercise on the object [10]. In distributed systems the capability object should be protected from modification and forgery through the use of cryptographically-secure hardware or software.

Capabilities are a particularly attractive solution to the problem of access control on parallel file systems because they have minimal impact on the scalability of the file system. Capabilities are sufficient for a server to authorize a file system operation on one of the resources contained in the capability. By using capabilities the I/O servers can perform authorization checks while continuing to operate independently of the other servers in the file system.

1.4 Statement of Research

Existing parallel file systems do not place a great enough emphasis on robust security for fear it will negatively impact the scalability and performance of the system. However, current trends towards the construction of large, consolidated compute clusters shared by a diverse group of users signify that security must play a larger role in parallel file system design. *We therefore present a robust capability-based security protocol for parallel file systems, and we demonstrate its performance*

and scalability properties with a reference implementation for the Parallel Virtual File System.

1.4.1 Target and assumptions

Our capability-based security protocol is designed to target a specific subset of parallel file systems running under a somewhat strict operating environment. We target user-level parallel file systems that run without any kernel protection other than address space separation and basic file system protections. Additionally, we assume that both the physical machine and the operating system kernel are secure from attackers. In particular we reject the possibility that network traffic can be intercepted by any process other than the one to which the traffic was directed, as this would require either physical access to the network or the ability to gain administrative privileges on a node.

1.4.2 Secure I/O and metadata operations

A file system's two most critical resources are its metadata and its data. File system metadata contains file attributes such as file size, timestamps, and permissions. File system data refers to the user-defined byte structure modified with *read* and *write* operations. We refer to the various data and metadata structures in the file system collectively as the *objects* of the file system, but we do so without the requirement that the file store be object-based.

In the interest of protecting file system objects from unauthorized access and modification we present a capability-based security protocol that works to eliminate the inherent security complications created by the decoupling of I/O servers that store a file's data from metadata servers that store a file's access attributes. The protocol requires clients to first request a capability for a file from a metadata server before attempting to access the file data on an I/O server. The capability is passed unmodified to the I/O server which can then perform authorization of the request based solely on the contents of the capability.

In our protocol the capabilities themselves are protected from modification and forgery by the addition of a *digital signature* to the capability. Public-key encryption is used to sign the capabilities in such a way that any capability not generated entirely by a trusted server will fail verification. Our reference implementation uses the widely-known RSA algorithm [17] and a static key configuration, but our protocol specification is sufficiently general to allow other algorithms and

configurations to be implemented as necessary.

1.4.3 Minimal scalability/performance impact

In the domain of high-performance parallel file systems a robust security protocol that sacrifices the performance or scalability of the file system is unlikely to be of much practical use. For this reason our protocol is designed to have little overhead while at the same time have the possibility to scale to large file systems with multiple metadata and I/O servers. Our protocol is based on capabilities because of their proved scalability properties.

1.5 Methodology

We begin by briefly reviewing the history of protection of computer resources. We pay particular attention to the implementation of early capability architectures. We then present a security protocol based on previous implementations of capability-based security that provides adequate protection from attackers while not overly limiting the scalability or performance of the file system.

In order to quantify the performance properties of our security protocol we provide a proof-of-concept implementation for the Parallel Virtual File System (PVFS) [3], a research parallel file system developed by Clemson University and the University of Chicago. As an Open Source research file system that is also used in production environments, PVFS provides us with the ability to prototype our code and test it against production hardware. Implementing the protocol as an extension to PVFS also provides us with the ability to perform in-depth performance comparisons between the original and security-hardened versions.

In Chapter 2 we outline related work on protection, distributed file systems, and security. In Chapter 3 we describe in detail our security protocol and how its prototype is implemented for PVFS. In Chapter 4 we detail our test framework and provide experimental results. In Chapter **??** we evaluate those experimental results with respect to our goals of performance and scalability. Finally, in Chapter 5 we conclude by summarizing our results and proposing several avenues for future improvements to the security protocol and implementation.

Chapter 2

Background and Related Work

Influences for our capability-based security protocol for parallel file systems span over 30 years of accumulated academic research on topics such as security, operating systems, cryptography, and distributed computing. We draw on early work concerning capability-based protection and combine it with recent advances in cryptography to meet the performance demands of modern parallel file systems. The resulting implementation is capable of scaling to future systems..

2.1 Protection of Machine Resources

The need to protect access to shared machine resources was recognized early in computing history. Early protection systems were commonly concerned with isolating the memory of concurrently running programs from each other [7]. Several contemporary protection schemes relied on hardware-enforced memory segmentation; however, the specifications of those schemes lacked a consistent jargon [12].

2.1.1 Lampson's access matrix

Seeing the advantage of a general mechanism for describing protection, Lampson devised the concept of an access matrix to formally describe the access attributes of a process with respect to an object in the system [12]. An access matrix consists of three essential components: *domains*, *objects*, and *access attributes*. Domains are the entities that have access to objects. In UNIX™ systems the domains correspond to processes and their corresponding effective user credentials. Objects are the

resources in the system being protected. For our purposes the objects are entities in the file system identified by their unique 64-bit handle value. Finally, the access attributes are a set of operations that can be performed on objects by domains.

The access matrix restricts the operations a domain can perform on an object. Element $a_{d,x}$ of an access matrix A specifies the access that domain d has to object x. Access is described in the form of a set of access attributes. For example, the access attributes for a file system object may consist of *read*, *write*, and *execute*.

From a theoretical standpoint the access matrix is an m-by-n matrix where m is the number of domains (e.g. users) and n is the number of objects (e.g. files). It is clearly infeasible to store the entire access matrix in memory as it may contain many billions of elements. Instead, common methods of representing the access matrix in memory involve taking into account the sparsity of the matrix and storing entries with an emphasis on either the columns or the rows of the matrix.

One convenient column-centric representation of the access matrix involves attaching an array of $(x, a_{d,x})$ pairs to the domain. Each pair maps an object (identified by its handle, x) to an access list. Lampson calls this pair a *capability*, and it is indeed quite similar to the capability objects used in our implementation. Capabilities are grouped into capability lists and stored with each domain. The system authorizes access by searching the capability list of the domain for the object x being acted upon. Capabilities are assumed to be protected from modification by hardware.

The alternative row-centric representation of the access matrix involves storing with each object an array of $(d, a_{d,x})$ pairs. Each pair maps a domain to an access list. The system authorizes access by searching the object for the domain requesting access. The row-centric representation of the access matrix is in fact an access control list (ACL).

2.1.2 Capability machine architectures

The earliest direct use of capabilities for resource protection was in machines whose hardware was specifically designed to take advantage of them. Examples of capability machines include the Cambridge CAP computer, the Flex machine, the Intel iAPX 432, KeyKOS, the Plessey System 250, and the IBM System/38 [13, 24, 16]. Each of the aforementioned machines generates capabilities for segments of memory identified by a $(\text{base}, \text{length})$ pair. Each of the machines also uses special hardware to enforce protection of the capability objects and prevent unauthorized access.

An important difference in the capability machines lies with how they prevent the forging of

capabilities. The Cambridge CAP, Intel iAPX, and the Plessey System 250 partition capabilities in special areas of memory to prevent their modification by unauthorized programs. The Flex machine, the IBM System/38, and KeyKOS implement a memory tagging scheme to allow capabilities to mix freely with data yet still be identified as protected objects.

2.1.3 Validation of objects after migration

When a protected resource is stored in virtual memory the operating system kernel is able to control access to that resource with the aid of the memory management unit. However, when that same resource is migrated to offline storage or transferred over the network the operating system must relinquish control. In response to the object migration issue Gligor and Lindsay developed a mechanism to allow the operating system to validate a resource coming from unprotected storage [8].

Gligor and Lindsay introduce the concept of a type manager that is responsible for maintaining positive control over its object representations to prevent the representations from being accessed or modified by untrustworthy subsystems. They present a mechanism for the type manager to *sign* object representations before they are transmitted outside of the control of the type manager, and in such a way that the type manager is able to later verify the signature with certainty. The object signatures are protected from forgery through the use of cryptography.

Two migrated object representations are discussed in the article. Both schemes require first signing objects by adding redundant bits to the external object representation. In the first object migration scheme the migrated object representation is formed by encrypting the entire external object representation, making it very difficult to forge a migrated object without knowing the private encryption key. The first scheme has the added benefit of preventing the object from being understood by anyone without access to the private key. When entities other than the type manager need read access to the migrated object the second scheme can be used to keep the external representation in the clear. In this scheme the signature is cryptographically derived from and stored with the external object representation. Only the actual signature bits are enciphered.

2.1.4 Capabilities in distributed systems

With the advent of distributed computing systems it became necessary to protect capability objects as they traversed a potentially hostile network environment. Hardware protection schemes used specialized host adapters to ensure the integrity of messages passing over the network. Software-based protection schemes typically relied on protection through either public-key encryption or a hash-based message encryption code.

One early system that employed distributed capabilities was the Amoeba distributed operating system described by Tanenbaum, Mullender, and van Renesse in [22]. The Amoeba system was notable for allowing user-level processes to directly manipulate capabilities in their own address spaces. Capabilities are protected cryptographically using special network hardware, although the authors admit that protection could also be achieved in software using public-key encryption.

2.2 Distributed File Systems

2.2.1 NFS

An early and widely used example of a distributed file system is the Sun Network File System (NFS). The goal of of the designers of NFS was to create a machine and operating-system independent method of transparently sharing file system resources over a computer network [18]. Transparent access is understood as the ability of clients to use the same set of system calls on local files as remote files. In order to provide for transparent access across a wide variety of machines the NFS specification [21] defines a protocol and not a specific implementation.

The NFS protocol is built on top of a remote procedure call mechanism where clients send requests to be processed by servers. The NFS protocol is completely stateless, meaning that the parameters to each procedure call contain all of the information necessary to process the call. A stateless protocol simplifies error recovery since clients and servers can simply be restarted whenever an error occurs.

The earliest versions of NFS were simple and convenient; however, they lacked any sort of robust security protection. For the most part security in NFS is purely advisory: an NFS client communicates its credentials to the server with every request, and there is no way for the server to ensure that the credentials have not been spoofed. The only real security checks in NFS are done

when the file system is first mounted and the server verifies that the request is coming from a list of approved clients. However, if one of the white-listed clients is compromised then that client can impersonate any user. The security holes inherent in the design of NFS were modified in the newest version of the protocol, NFSv4, which mandates strong security that is supported through the use of cryptography and stateful protocols [19].

2.3 The Parallel Virtual File System

The Parallel Virtual File System (PVFS) is a parallel file system designed to provide scalable, high-performance access to data for parallel scientific computing applications [3]. PVFS uses a client/server model where clients send requests to servers who then process the requests and return responses. Servers are split into two categories: metadata servers and I/O servers. Metadata servers handle the file system metadata that contain attributes such as file sizes, timestamps, and permissions. I/O servers are responsible for the bulk of data transfers to files. Although metadata servers and I/O servers are logically separated it is perfectly possible for both servers to run in the same process.

Scalability of the file system is achieved through careful design of the communication paths between clients and servers. In the design of PVFS there was an emphasis on minimizing shared state [3]. Much like early versions of NFS the PVFS server does not contain any distributed locking mechanism and the client request protocol is stateless [18]. As a practical result of the stateless design each request must be fully self-contained. While a stateless request protocol greatly improves scalability it requires careful planning of any new request protocols to ensure the server does not depend on the results of previous requests.

Chapter 3

Design and Implementation

3.1 Structure of a Capability

After deciding that capabilities were a satisfactory solution to our security and scalability needs we set out to design a capability structure suitable for use in our protocol. Our design was motivated by three overarching requirements:

1. **that the migrated capability object be protected from spoofing.** A capability object that can be modified by malicious clients provides no real security.

2. **that the capability contain enough information to validate any request.** Scalability of a parallel file system requires that as little state be kept on the servers as possible. Communication between servers should also be kept to a minimum. A capability should be the only object required to perform authorization of basic file system operations.

3. **that the number of capability objects required for a file system operation be as small as possible relative to the size and layout of the file.** Previous capability implementations for file systems have required as many as one capability per block [9]. If possible we would like a single capability to cover an entire file in order to minimize the number of capabilities passing over the network.

Our design is also based on some basic assumptions about the computing environment. We assume both client and server nodes are under the control of a system administrator and that this

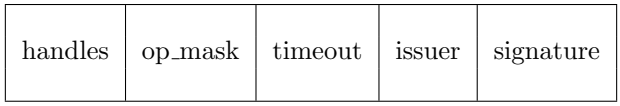

Figure 3.1: Fields of a capability

administrator is the only user with the ability to configure the operating system of the nodes. We also assume a local area network connects clients and servers and that attackers are not able to sniff or inject packets on this network.

The reason we make these assumptions is to simplify the design of the capability protocol for use with PVFS, which has a stateless, and possibly connectionless, communications protocol. A stateless networking protocol means we are not able to make use of an encrypted channel, and thus we are not able to prevent against replay attacks where a capability is sniffed off the network and used by an unauthorized client. By ensuring that both the network and the nodes are operating in a secure environment we create a situation where an attacker is almost surely unable to sniff capability objects, because this would require either physical access to the computer network or administrative-level access to a node.

Based on the above requirements and assumptions we present the capability object shown in Figure 3.1. Descriptions of the various fields of the capability object are provided in subsequent subsections.

3.1.1 Target handles

The *handles* field is a collection of all the handles to which the capability applies. Before a server permits a request to proceed it first ensures that the handle of the object being operated upon is contained in *handles*. In PVFS this array contains the metadata handle and each of the data handles of a file. Storing multiple handles in a single capability is an optimization that avoids the creation of an excessive number of capability objects, as the number of handles for a file is typically one more than the number of servers. The *handles* field is similar to the *object* field in an Amoeba capability [22].

Symbol	Permission granted
PINT_CAP_EXEC	Execute a file; traverse a directory
PINT_CAP_WRITE	Write to a file; change the contents of a directory
PINT_CAP_READ	Read from a file; list the contents of a directory
PINT_CAP_SETATTR	Set file/directory attributes
PINT_CAP_CREATE	Create a new object
PINT_CAP_ADMIN	Perform administrative-level functions
PINT_CAP_BATCH_CREATE	Create multiple new objects
PINT_CAP_BATCH_REMOVE	Remove multiple objects

Table 3.1: Bits in the capability mask

3.1.2 Permitted operations mask

By definition a capability must have some list of permitted operations associated with the capability. In our capability the *op_mask* field is a bitmask of operations the owner of a capability is allowed to perform on the target handles. A '1' in a bit position signifies that the corresponding operation is permitted. All currently defined operations are listed in Table 3.1.

The PINT_CAP_READ, PINT_CAP_WRITE, and PINT_CAP_EXEC bits have meanings identical to the *read, write,* and *execute* bits of traditional UNIX™ file systems as specified in [1]. The PINT_CAP_SETATTR bit encapsulates the ability to modify the attributes of an object; in POSIX file systems this permission is granted only to the file or directory's owner [2]. Before a user can create an object in the file system he must have a capability for a directory with the PINT_CAP_CREATE bit set. This permission is a workaround for the fact that creating a file and adding it to a directory are two separate server operations in PVFS.

The final three operations are less frequently used. The PINT_CAP_ADMIN bit is set for certain administrative operations that modify the configuration or internal state of the file system. The PINT_CAP_BATCH_CREATE and PINT_CAP_BATCH_REMOVE operations are used internally by the file system server when creating or destroying large numbers of files.

3.1.3 Capability timeout

The purpose of the *timeout* field is to limit the amount of time a capability can be used. The field is set to the point in time when a capability is no longer valid. Part of the capability verification process is ensuring the capability has not timed out; if the current server time is greater than or equal to a capability's *timeout* value then that capability fails verification.

Since a capability is the primary object underlying authorization of server requests the timeout can be understood as an upper bound on the time between when permissions on a file system object change and when a client will notice those changes. The capability timeout is the only means of revoking a capability in our protocol.

3.1.4 Digital signature

The *signature* field contains a digital signature of the capability to allow a server to verify that the capability was created by a trusted entity and not subsequently modified. When a server generates a capability it places its own name in the *issuer* field so that later servers will know what public key to use when verifying the digital signature.

3.2 Capability Protocols

After finalizing the form of a capability object we proceeded to focus on its function. The way that capabilities are used in a parallel file system is defined by the protocols that support them. We modify the request-response protocol of typical parallel file systems to support capability-based security by adding a capability to the beginning of every request. Parallel file system servers are then able to authorize requests based solely upon the fields of the capability. Before a capability can be used, however, it must first be generated.

3.2.1 Capability generation algorithm

When a client wishes to perform an operation on a file system object it first issues a capability generation request to the metadata server that owns the object. The request contains the authentication credentials of the client and the handle of the object in question. Object handles are normally obtained as the result of a *lookup* request on the object's parent directory.

Upon receiving a request to generate a capability the server looks up the attributes of the target object in its local database. It uses the credential information in the request along with the access control list contained in the object's attributes to generate an effective permissions mask. This mask is stored in the *op_mask* field of a newly generated capability and represents all of the operations the client is allowed to perform on the target object. The *handles* field of the capability

is initialized to the handle of the target object, and as an optimization if the object is a regular file then the corresponding datafile handles are also added.

After initializing the *handles* and *op_mask* fields of a new capability the server proceeds to sign the capability. Before signing the capability the server places its name into the *issuer* field and sets the *timeout* field to the sum of the current time and some configurable constant value. At this point the server can calculate a hash of the capability, encrypt the hash with its private key, and store the result in the *signature* field.

The result of the previous operations is a fully-formed capability representing the rights of the client for an object in the file system. The server returns this capability to the client who can then use the capability in subsequent requests that access the original target object.

3.2.2 Capability verification algorithm

Capability verification is an important operation that occurs at the beginning of server request processing. Clients attach a capability to the beginning of every request. Because the capability is often the only object used to authorize requests the server must first verify its authenticity to ensure it has not been modified or forged. The process of verification involves both inspecting the fields of the capability to ensure they contain correct values and validating the digital signature attached to the capability.

The signature validation algorithm first examines the *timeout* field of the capability to ensure the capability remains valid. Every valid capability will have a value in *timeout* that is strictly greater than the current server time. Next the public key corresponding to the private key used to sign the capability is loaded from the trust store indexed by the capability's *issuer* field. Finally, the digital signature is verified using the issuer's public key. If the issuer's public key cannot be located in the database it signifies that the issuer is not trusted and the capability fails verification.

A notable exception to the validation algorithm occurs when the capability being validated is the *null capability*. The null capability is sent with requests for which a capability is not appropriate. An example is the PVFS `get-config` request that requests a copy of the server's configuration file. The null capability does not represent a security risk because its *op_mask* field contains all zeros, meaning that it will never pass a permission check.

After validating the cryptographic integrity of the capability the verification code examines more closely the fields inside the capability. The *handles* field is checked first to ensure that the

handle of the request's target object is covered by the capability. The request types that don't have a target object ignore *handles*. Next, a bitmask specific to the type of request is generated and compared to the *op_mask* field to ensure that the client has permission to operate on the target object. For example, a write request processor will check for the presence of the `PINT_CAP_WRITE` bit in *op_mask*. Request processors expecting a null capability ignore the value of *op_mask*.

If a capability passes these initial verification checks it is considered valid and the request-specific state machine is invoked to begin processing of the request. Certain requests may perform additional checks on the capability inside their respective state machines.

3.3 Structure of a Credential

In order to initially generate a capability for a client the server must have some knowledge of the client's user credentials so that it can locate the relevant entry in its access control list. Our protocol defines an object that we call a *credential* to assist the client in identifying itself to a server. Authentication of users across machine boundaries is a complex issue, and its implementation is often operating-system dependent. Our goal in designing the credential object was primarily to simplify the implementation of capabilities for parallel file systems running in a UNIX environment.

In keeping with our simplistic approach towards client authentication we make several assumptions about the operating environment of the client and server nodes of the parallel file system. We assume that the nodes are running a Unix-like operating system that identifies users by a combination of their user identification number and some nonempty set of group identification numbers. Additionally, we assume that the client and server nodes are all part of the same authentication domain, such that a given user identification number identifies the same user on any node in the file system group. Typical clusters will implement a single administrative domain using a tool such as the Network Information Service [20].

Based on the previous requirements and assumptions we present the credential object as having the structure described in Figure 3.2. Our design is intentionally simplistic, offering a thin wrapper around the user credentials usually provided to a process by the operating system kernel. Detailed descriptions for the fields of a credential are provided in subsequent subsections.

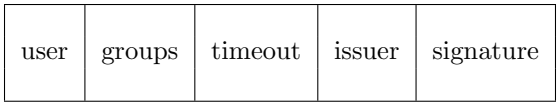

Figure 3.2: Fields of a credential

3.3.1 Identifying a user

The *user* field contains the integer user identification number that uniquely identifies a single user to any node in the file system group. The *groups* field is a nonempty set of group ids of groups of which the user is a member. The first member of *groups* is the primary group id of the user.

The *user* field is typically set to the effective user id [1] of the client process accessing the file system. The *groups* field normally contains a list of all the groups corresponding to *user*. Secondary groups are included in the credential to facilitate capability generation for files whose group is set to one of the secondary group ids.

3.3.2 Credential timeout

The purpose of the *timeout* field is to limit the amount of time a credential can be used. The field is set to the point in time when a credential is no longer valid. Part of the credential verification process is ensuring the credential has not timed out; if the current server time is greater than or equal to a credential's *timeout* value then that credential fails verification.

The credential timeout is the primary means of ensuring the integrity of the system in the event that a user's account is modified. Were it not for the credential timeout, a user could store a credential and use it indefinitely. In effect the credential timeout places an upper bound on the amount of time between when a user's account is modified and when the server enforces this modification.

3.3.3 Digital signature

The *signature* field contains a digital signature of the credential to allow a server to verify that the credential was created by a trusted entity and not subsequently modified. When a server generates a credential it places its own name in the *issuer* field so that later servers will know what public key to use when verifying the digital signature.

3.4 Credential Protocols

With the basic structure of a credential defined we were free to design its integration into the usual parallel file system protocols. Unlike a capability that is sent in the header of every file system request, the credential is only needed for certain requests that require knowledge of a user's identity. As such our protocol allows a credential to be sent in the operation-specific part of a request and only verified when necessary. Servers are responsible for knowing when a credential is attached to a request and verifying it accordingly. For example, a request that causes a file to be created in the file system will need to know the identity of the user creating the file so that it can set the file's initial ownership attributes. In our protocol the servers do not generate credentials; rather, they simplify verify the credentials sent from clients.

3.4.1 Credential generation algorithm

The credential generation algorithm is highly implementation-dependent. In most cases we envision credential generation being performed by the operating system or by a privileged utility running on the client node. User code should be able to call this routine as a service to generate a credential for the effective user id of the current process. The client may then send the credential as a part of any file system requests that require one.

3.4.2 Credential verification algorithm

Verification of a credential involves only those steps necessary to ensure the credential has not been modified or forged. The server currently has no concept of a "valid" user identifier. It can only verify a credential based on the trust it places in the entity that signed the credential.

Validation of credentials is performed in the same way as validation of capabilities as explained in Section 3.2.2. First, the *timeout* field is checked to ensure its value is strictly greater than the current server time. Next, the public key for the credential's issuer is loaded from the trust store based on the value of the *issuer* field. Finally, the digital signature of the credential is verified against the issuer's public key. If a credential passes all of these tests it is considered valid.

3.5 Reference Implementation

In order to verify the correctness of our capability protocol and test its performance we developed a reference implementation for the Parallel Virtual File System (PVFS). The source code to PVFS is licensed under an Open Source license, and owing to the modularity of its design we were able to modify the client request protocol to incorporate capabilities. The bulk of the security implementation was added as a library for use by the rest of the source tree. The final part of the implementation of capability-based security for PVFS was modifying each server request-processing state machine to perform security checks relevant to the type of request. Because the specific changes to this part of the codebase are so numerous we include them in the Appendix.

3.5.1 Security in vanilla PVFS

The unmodified version of PVFS makes a best-effort attempt at securing file system operations. However, in most cases these checks implement a purely advisory security model. The source of the difficulty in performing robust security checks in vanilla PVFS is a result of the disconnect between the I/O servers that process that data and the metadata servers that contain the security attributes. This disconnect means that an I/O server has no way of ensuring that a client has permission to modify a data file. For this reason many security checks are done in the PVFS client library that implements the canonical request protocol. However, there is no mechanism to stop an attacker from modifying the client library to remove such access checks.

3.5.2 Security support library

We began our implementation of robust security for PVFS by building a library of common security routines whose use we anticipated throughout the file system code. The support library contains the code to generate and verify capabilities and credentials. We also include the handling of several configuration options specific to our security implementation.

The capability and credential code is based on two configurable quantities: the server private key and the trust database. Each server is configured with its own private key to use when signing capabilities. This private key should be stored in a secure location in the server's local file system to prevent an unauthorized process from impersonating the server. The trust database is a collection of the public keys of every server and client in the file system indexed by name. When a server

wishes to verify a capability or credential it will use the *issuer* field to locate the correct public key in the trust database. We currently read the trust database from disk and store it in a hash table to support quick lookups.

3.5.3 Enhancements to the server code

A great deal of changes were made to the PVFS server code to support capability-based security. The first part of the code modified was the request protocol. We modified the protocol to include a capability object in every request and a credential object in those requests that require knowledge of the identity of the client. We the modified the server request-processing state machines to verify and make use of the capabilities and credentials.

When a request arrives over the network from a client the PVFS server determines the correct server state machine to run based on the type of request. Before the request-specific state machine is invoked a special *prelude* state machine runs to perform the processing tasks common to every request. It was in this prelude state machine that we inserted code to verify capabilities and credentials according the algorithms given in Section 3.2.2 and Section 3.4.2. If either a capability or a credential fails verification then request processing stops immediately and an error is returned to the client. In the event that the capability and/or credential passes verification the request-specific state machine is invoked in the normal fashion.

The bodies of certain request-specific state machines were modified to support capability-based security. While the full details are too complex to provide here (we refer the reader to the Appendix), we will briefly mention the most important modification to the `get-attr` state machine.

The `get-attr` state machine is normally tasked with returning the attributes of a file system object given its handle. Because of its association with object attributes and the fact that reading the attributes of a file system object is permitted regardless of the permissions of that object [2] we chose to implement capability generation in this state machine. Client code will request a capability for an object by setting a special bit in the `get-attr` request and sending its credentials as a part of the message. The server will then use those credentials to generate and return a capability for the client.

3.5.4 Enhancements to the client code

The most common form of client code is the PVFS system interface that is implemented as a library that wraps certain file system primitives like creating a directory, writing to a file, or changing the attributes of an object. Much like the server code, the PVFS system interface is designed using state machines that break a large request into smaller pieces. The addition of capability-based security to the PVFS protocol changed the semantics of certain requests significantly, and this required a corresponding change in the client state machines. The vast majority of changes involve performing an extra `get-attr` request before attempting to read or modify a file system object. We provide a complete list of changes in the Appendix.

An example of a typical client state machine is the `io` state machine that reads and writes file data. An example of a read request is given in the sequence diagram of Figure 3.3. When the I/O state machine is executed it is given as an argument the handle of the file to read from. Before it can perform I/O on the file, however, it must first acquire a capability that covers the file's handle. To acquire the capability the client contacts the metadata server responsible for the file and asks it to generate a capability. The metadata server returns a capability that the client then uses to send a read request to the I/O server that contains the file data. The I/O server uses the capability to verify that the client has read access to the file before it returns any data.

A small number of client operations become considerably more complex after implementing capability-based security. An example is the `lookup` state machine that looks up a named entry in a directory. The lookup state machine is used to find the handle of a file or directory based on its path. While the lookup operation is conceptually simple, its secure implementation is made more complex by the requirement that the client acquire a capability for each directory in the path before performing a lookup on it.

An example of a lookup operation is given in 3.4. In PVFS every lookup starts at the root directory and continues down. The root handle is well-known, so the first operation of the lookup state machine is to request a capability for the file system root. The client lookup state machine then continues to walk the path until the handle for the final path element is located..

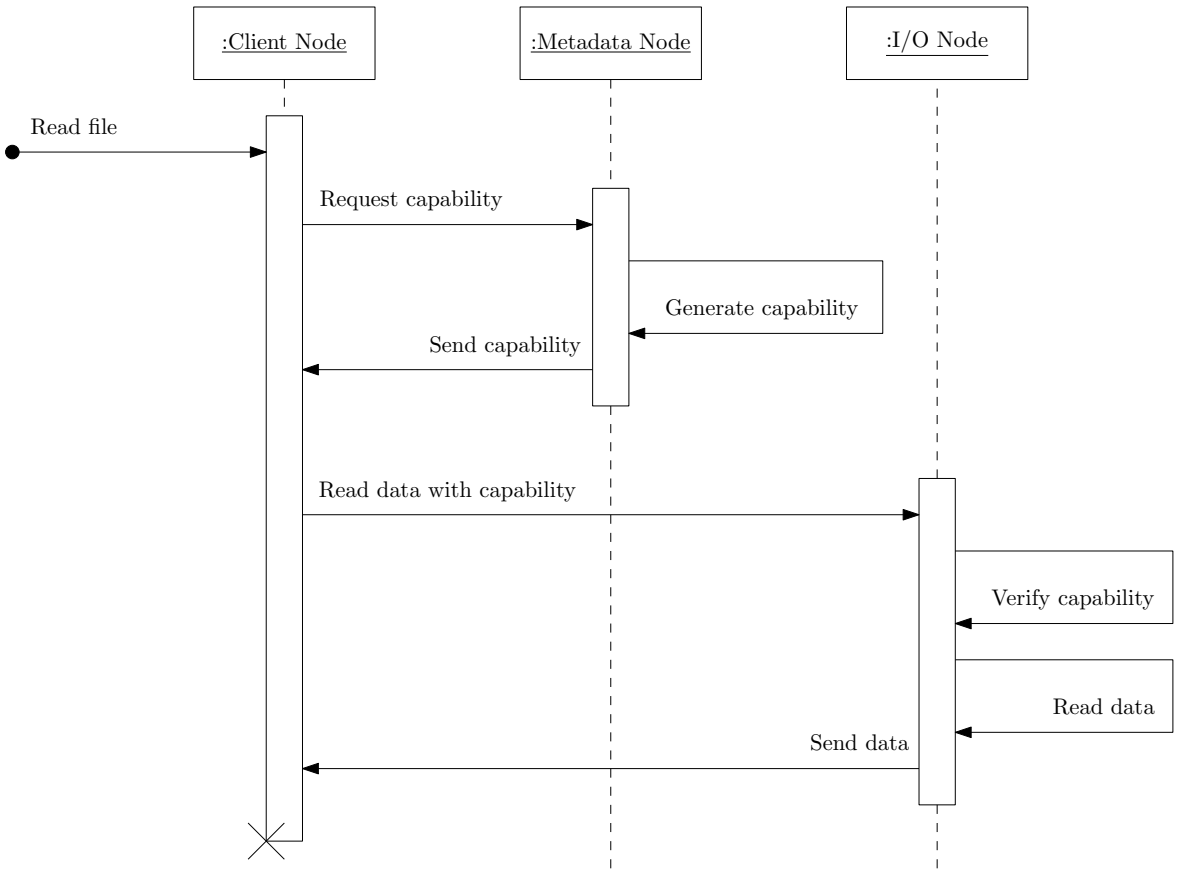

Figure 3.3: Read sequence diagram

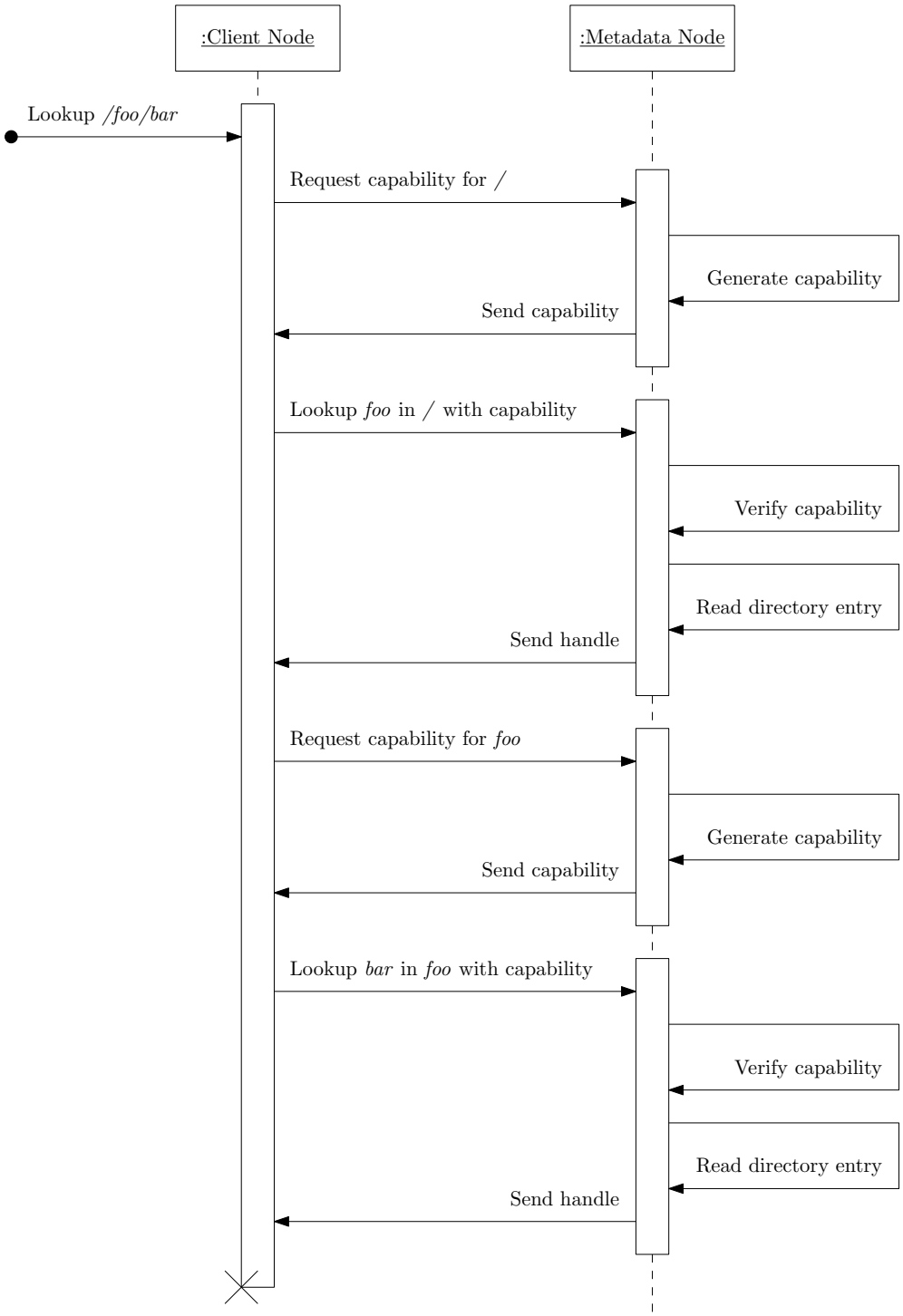

Figure 3.4: Lookup sequence diagram

3.5.4.1 Client generation of credentials

The code added to the client to generate credentials bears special mention. The generation and management of user credentials is essentially an administrative function based on an organization's security policy. To keep our implementation as general as possible we provide a rudimentary yet flexible method for generating credentials in the PVFS client library.

Consumers of the PVFS client library must pass a credential object to every file system function. This credential object is sent to the server unmodified for those requests that require knowledge of a user's identity. To simplify the process of generating credentials we provide a set of utility functions and an administrative application named `pvfs2-gencred` that relies on the operating system kernel to provide credential information.

The default administrative application uses a system call to obtain the user credentials of the process that spawned it. It then uses several more system calls to fill in the primary and secondary groups associated with that user account. As a final step it signs the credential with its private key. The corresponding public key must be installed in every server's trust database for this method to be effective.

Credential generation is performed in a separate process for security purposes. The `pvfs2-gencred` application is intended to be installed setuid the user with read access to the private key used to sign credentials. By executing the setuid `pvfs2-gencred` application the client process can generate a credential without itself having permission to read the private key file. The client utility functions take advantage of this fact by performing a fork-and-exec of `pvfs2-gencred`, which generates a credential for the client application and passes it back via a pipe [1]. The `pvfs2-gencred` included in the reference implementation is intended as a starting point for more complex authentication policies developed by a system administrator.

Chapter 4

Experimental Results

4.1 Experimental Setup

Experiments to test the performance and scalability of our security protocol were run on a small commodity cluster with nodes containing dual Intel Xeon 3040 processors and two gigabytes of random access memory. The Xeon 3040 implements the Core micro-architecture and is clocked at 1.87 GHz. Each of the nodes runs the 64-bit CentOS 5.5 Linux distribution and is installed with OpenSSL version 0.9.8e-fips-rhel5. The nodes are connected with gigabit Ethernet.

There are three versions of the PVFS client and server software installed on each node. The first version is the latest repository version of PVFS. We refer to this version as *Orange*. The second version is identical to Orange except it implements our robust security protocol. We refer to this version simply as *Security*. The final version, *Nosec*, is identical to Security except its cryptographic primitives have been replaced with no-ops.

Having three different versions of the PVFS software installed allows us to perform meaningful tests to measure the performance overhead of our security protocol. The version of PVFS without any extra security checks, Orange, serves as the baseline for any measure of overhead. We compare Orange to Security using several different microbenchmarks designed to stress the slowest parts of the Security implementation. We also run each microbenchmark with the Nosec version of PVFS to measure the overhead due solely to extra requests performed as a part of the security protocol.

The microbenchmark used to measure performance overhead and scalability is a heavily

modified version of a tool we call *Diskperf*. The original Diskperf is a simple tool that creates, writes, and reads files using the normal system calls. The modified Diskperf used in our benchmarks makes direct use of the PVFS client library. Using the client library directly enables us to avoid the overhead of the PVFS kernel module.

For the most part each of our PVFS servers is configured using the default configuration from the `pvfs2-genconfig` tool. The only exception is when we run the metadata performance benchmarks. In these benchmarks we wish to measure the overhead of metadata operations without taking into account the latency of writing data to disk. For these tests we use the `null-aio` configuration option on the PVFS server to cause the server to discard any data that is written and return zeros whenever data is read. This mode only applies to file data. Metadata is still written to persistent storage to ensure the file system remains in a consistent state.

4.2 The Cost of Cryptography

The capability objects used in our protocol are protected from modification and forgery by affixing a digital signature to the objects. Our protocol employs public key cryptography to generate the digital signatures instead of the faster symmetric key algorithms. The result of this decision is that a significant amount of request processing time is spent generating secure hashes and performing encryption operations.

We devised a test to measure exactly how much time is spent performing cryptographic operations. This test was built by moving the capability signing and verification code to a separate program and profiling it in a tight loop. We then generate keys of various sizes and measure how many sign and verify operations can be performed in one second. The results of this test are shown in Figure 4.1.

The results of this test reveal several interesting details about the RSA algorithm used to sign and verify capabilities. The first is that, as expected, the key size significantly affects the amount of time required to sign or verify a capability. Using our implementation it is really only practical to use key sizes of at most 1024 bits. The second detail is that it takes must less time to verify a capability than to sign one: in the worst case with 4096 bit keys the server can perform nearly 64 times more verifying operations than signing operations. This detail would suggest that we should attempt to minimize the number of times a capability is generated relative to the number

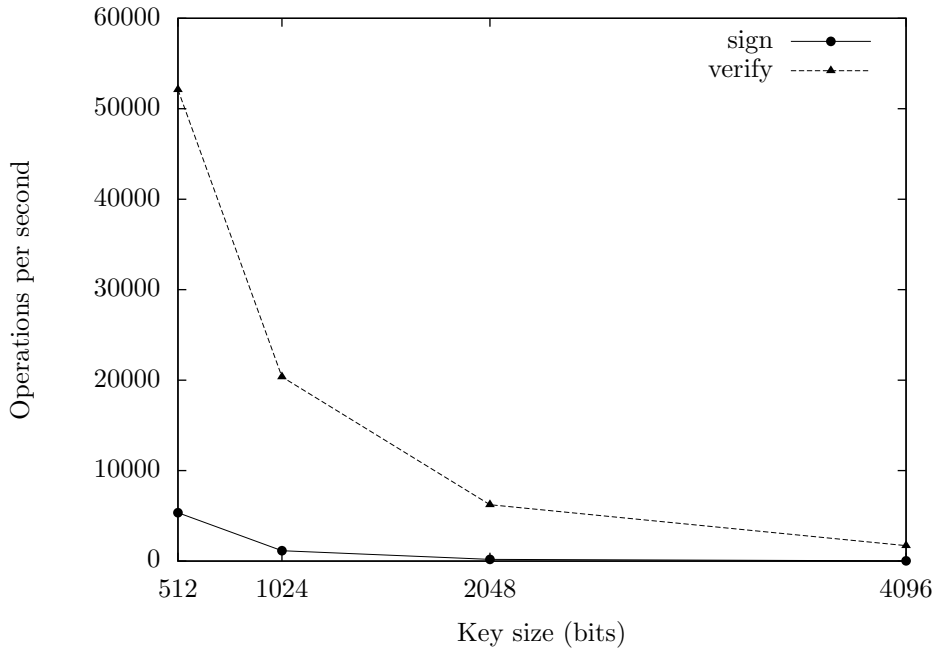

Figure 4.1: Capability operation times

of times it is used.

4.3 Performance Overhead of Metadata Operations

The amount of overhead introduced by our security protocol is proportional to the number of requests being processed. Metadata-intensive workloads create a large number of requests and are therefore significantly slowed by the time spent signing and verifying capabilities. An example of a metadata-intensive workload is the mpiBLAST program used to compare DNA sequences [5].

The mpiBLAST program creates a large number of small files. In order to simulate this type of metadata-intensive workload we configure the Diskperf tool to write and then read back large numbers of files using a single four kilobyte data transfer. The PVFS servers are placed in a special *null-aio* mode that causes file data to be thrown away instead of stored on disk. Reads performed when the server is in this mode cause the server to return all zero bytes. Placing the server in this mode allows us to minimize any latency hiding caused by writing to disk. The resulting performance data is influenced solely by how long the server spends performing metadata operations.

Results for the write tests are shown in Figure 4.2 and results for the read tests are shown

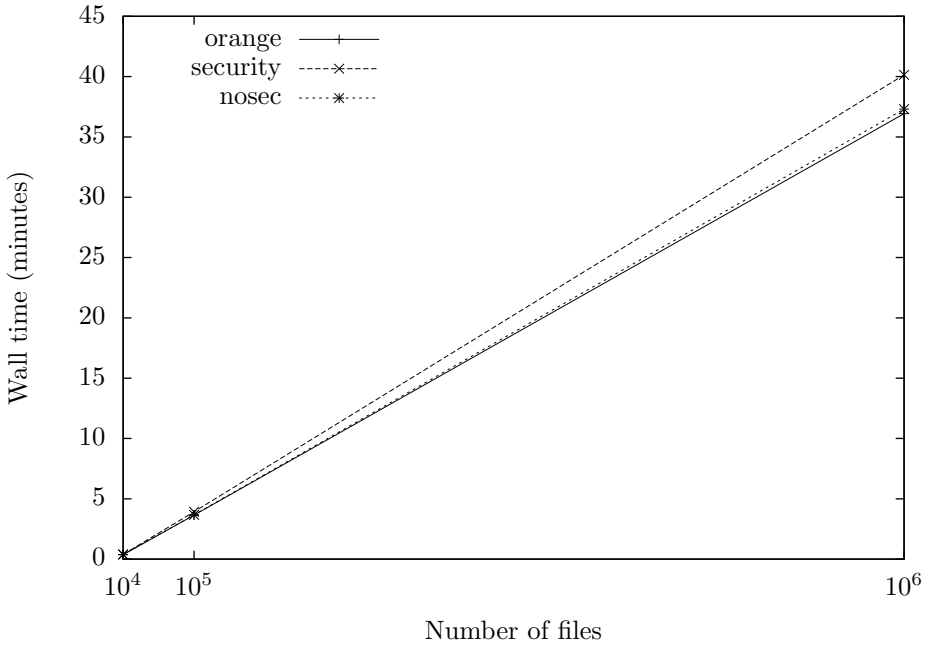

Figure 4.2: Writing small files

in Figure 4.3. Each test is performed with a single client node and four server nodes. We sweep the number of files from 10,000 to 1,000,000 and run the tests with all three server builds. The Security build uses 512 bit keys for capabilities and 1024 bit keys for credentials.

The results shown in Figure 4.2 for writing small files show that the Security build has a fairly low performance overhead when compared to the Orange build. In the worst case of one million files the overhead is less than nine percent. We also see that the Nosec branch with cryptographic operations disabled has only about one percent overhead when compared to Orange, which leads us to hypothesize that the performance overhead in the Security build is caused almost entirely by the time required to sign and verify the capabilities.

The results of small reads shown in Figure 4.3 are less than promising. The overhead of Security when compared to Orange is around 56% in this case. Even the overhead of Nosec is fairly high at 12%.

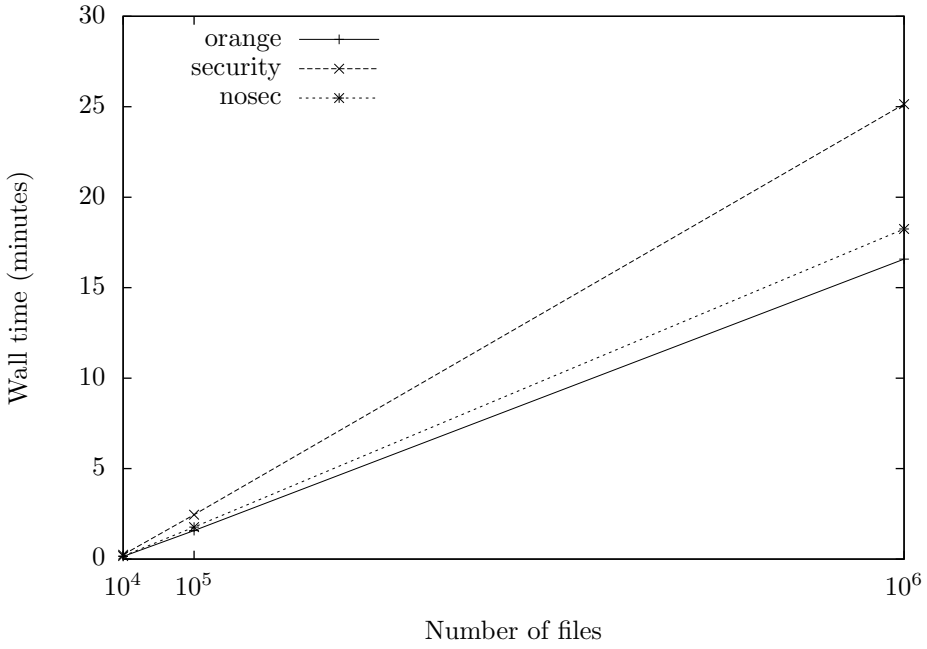

Figure 4.3: Reading small files

4.4 Scalability of Capability-based Security

More important than the performance overhead caused by using capability-security with a single client is the scalability of the system when serving requests from multiple clients simultaneously. To measure the scalability of our implementation we configured eight different nodes to use Diskperf to write files to the same directory simultaneously. The file system is again using 512 bit keys for capabilities and 1024 bit keys for credentials.

In the first test we use the use the `null-aio` backend combined with a blocksize of only four kilobytes to measure the scalability under metadata-intensive operation by writing 100,000 files with each of the eight clients. The results for this test are shown in Figure 4.4. The results show that while the system as a whole is not scalable, the Security version of the server does not exhibit poorer scalability than Orange.

In the second test we instead measure the scalability of the file system under an I/O intensive load. We configure the PVFS server with its default `alt-aio` disk backend and for the first time include the actual disk performance in our results. The eight Diskperf nodes each create a single two gigabyte file by writing 1 megabyte blocks. The results of this test are shown in Figure 4.5. The

results of this test stand in stark contrast to those of the first test. In fact, the scalability appears to be superlinear. We conclude that the scalability of the file system is exceptional when the number of I/O operations is very large compared to the number of metadata operations.

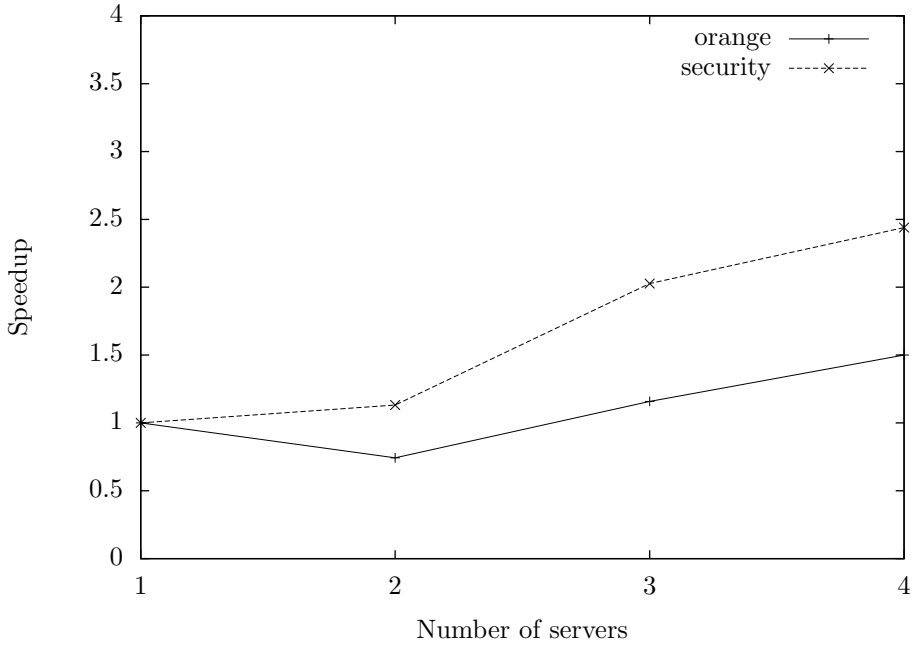

Figure 4.4: Scalability of metadata operations

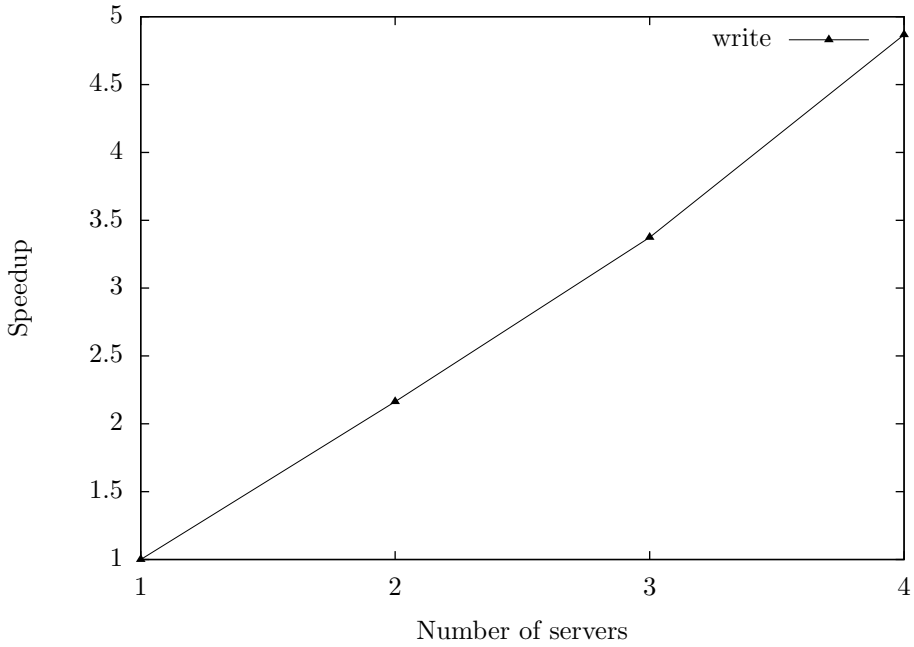

Figure 4.5: Scalability of I/O operations

Chapter 5

Conclusion

We presented a scalable capability-based security protocol suitable for use in high-performance parallel file systems. We based our design off of previous research into the use of capabilities in distributed systems. Our protocol diverges from earlier distributed capability protocols when necessary in order to meet the demanding performance requirements of the modern supercomputing cluster.

To verify and test our protocol we developed a reference implementation for the Parallel Virtual File System. We measured the performance and scalability of our implementation using several microbenchmarks designed to stress the weakest parts of the protocol. Test results showed that the performance and scalability of our implementation are highly dependent on the desired level of security and the type of file system accesses being performed. Thus, while our implementation proves that it is possible to solve the inherent security problems of parallel file systems using capability-based security, it demonstrates that the scalability and performance of the basic protocol does not meet the demands of metadata-intensive fie system workloads.

5.1 Future Work

Our test results demonstrate that there is much room for improvement in the area of scalability and performance of metadata-intensive file system workloads. We present several directions for future work to eliminate metadata bottlenecks.

5.1.1 Aggressive caching of capability objects

Test results for our implementation of the capability-based security protocol clearly indicate that the most significant bottleneck is the time spent signing and verifying capability and credential objects. We saw that the number of sign and verify operations per second drops precipitously for key sizes greater than 1024 bits. We also saw that metadata-intensive access patterns perform the worst because they generate a large number of requests, with each request requiring the verification of at least one capability.

Our idea for a solution to the metadata problem is to aggressively cache capabilities on both the server and the client. Clients would cache capabilities to avoid the round-trip message time of requesting a capability from the server when a suitable one already exists in the cache. Servers would cache capabilities to avoid the expense of verifying every capability. They would instead keep a cache of recently-verified capabilities with the expectation that a client will use the same capability multiple times. The assumption of course is that the amount of time required to locate an entry in the cache is far less than the amount of time required to perform a capability verification.

5.1.2 Higher performance of cryptographic operations

Besides caching another way to speed the operation of the file system under metadata-intensive access patterns would be to directly increase the speed of sign and verify operations. Increasing the speed could be done by experimenting with alternative cryptographic implementations, possibly ones implemented in hardware. The protocol could also be changed to use symmetric key encryption and hash-based message authentication codes in place of the much slower public-key algorithms. However, doing so will somewhat decrease the security of the system since a compromise of a single file system node would mean a compromise of the entire file system.

Chapter 6

Appendix

6.1 Modfications to Client State Machines

6.1.1 Modifications to every state machine

Modifications to each client state machine were neccessary to support the use of capabilities. The external interface of the client API was modified to accept the new credential object used to identify a client. This credential object is used internally by the state machines to request a capability before performing a server operation.

Every server operation now requires that a capability be included in the request. For most operations the capability is for the object being modified; however, certain file system operations do not have a target object. For these operations the special *null* capability is sent.

6.1.2 Client `mgmt-create-dirent` state machine

The `mgmt-create-dirent` state machine is a management state machine used by `pvfs2-fsck`, the file system check application. The state machine is a wrapper around the `CRDIRENT` request that creates an entry in a directory to point to an existing file, directory, or symlink. The major modification to this state machine was to request a capability for the parent directory before adding a new entry to it.

6.1.3 Client `mgmt-remove-dirent` state machine

The `mgmt-remove-dirent` state machine is a management state machine used primarily by file system repair routines to remove specific directory entries. The state machine is a wrapper around the `MGMT_REMOVE_DIRENT` request. The primary modification to this state machine was to request a capability for the parent directory before removing an entry from it.

6.1.4 Client `mgmt-setparam-list` state machine

The `mgmt-setparam-list` state machine is a management state machine used primarily by file system repair routines to set run-time parameters on a list of servers. The state machine is a wrapper around the `MGMT_SETPARAM` request. The primary modification to this state machine was to request a capability for the file system root, which an administrative user can then use to set server run-time parameters.

6.1.5 Client `sys-create` state machine

The purpose of the `sys-create` state machine is to create a new file in the file system. The client library code for file creation is somewhat complicated in that it must send separate requests to create the metadata file, the data files, and the directory entry for the new file. Modifications to this state machine were made to request a capability before each request is sent.

6.1.6 Client `sys-del-eattr` state machine

The `sys-del-eattr` state machine removes an extended attribute from an object by name. The primary modification to this state machine was to request a capability for the object being modified.

6.1.7 Client `sys-io` state machine

The `sys-io` state machine is the state machine used to read from and write to files. The primary modification to this state machine was to request a capability for the data files that are the targets of the I/O operation.

6.1.8 Client `sys-lookup` state machine

The purpose of the `sys-lookup` state machine is to lookup the handle of an object given its path in the file system. The original version of this state machine was optimized for the case when the metadata for multiple consecutive path elements were located on the same server. In the secure version, however, this optimization was removed because it is no longer possible to lookup multiple path elements. Instead a capability must be requested for each directory along the path.

6.1.9 Client `sys-mkdir` state machine

The purpose of the `sys-mkdir` state machine is to create new directories in the file system. The primary modification to this state machine was to request a capability for the parent directory of the directory being created.

6.1.10 Client `sys-remove` state machine

The `sys-remove` state machine deletes a logical file from the file system by removing its directory entry, data files, and metadata file. The primary modification to this state machine was to request a capability before each removal operation.

6.1.11 Client `sys-rename` state machine

The `sys-rename` state machine changes the path of a file, potentially moving it to a different directory. If the file is changing directories the `sys-rename` state machine removes the old directory entry and creates the new one, otherwise the state machine simply changes the name in the current directory. The primary modification to this state machine was to request a capability for each directory before it is modified.

6.1.12 Client `sys-setattr` state machine

The purpose of the `sys-setattr` state machine is to set the attributes of a file system object. The primary modification to this state machine was to request a capability for the object whose attributes will be modified.

6.1.13 Client `sys-set-eattr` state machine

The purpose of the `sys-set-eattr` state machine is to set the extended attributes of a file system object. The primary modification to this state machine was to request a capability for the object whose extended attributes will be modified.

6.1.14 Client `sys-small-io` state machine

The `sys-small-io` state machine handles reads and writes on small files. The primary modification to this state machine was to request a capability for the file on which the I/O will be performed.

6.1.15 Client `sys-symlink` state machine

The purpose of the `sys-symlink` state machine is to create symbolic links in the file system. A symbolic link is a special file that contains a reference to another file in the form of an absolute or relative path name[1]. The `sys-symlink` state machine was modified to request a capability for the newly created metadata file before setting its attributes and for the parent directory before creating a directory entry to the new file.

6.1.16 Client `sys-truncate` state machine

The purpose of the `sys-truncate` state machine is to resize a file. The primary modification to this state machine was to request a capability for the file being resized. A superfluous access check was also removed as the server is fully responsible for request authorization.

6.2 Modifications to Server State Machines

6.2.1 Modifications to every state machine

Certain modifications were made to every server state machine to support the new capability-based security mechanism. Each state machine is now responsible for supplying a function that can be called to verify that a capability contains the appropriate permissions.

6.2.2 Server `batch-create` state machine

The purpose of the `batch-create` state machine is to facilitate the creation of large numbers of file system objects. This state machine is used primarily by the handle preallocation code to reserve data files on servers before they are needed. The state machine expects a capability with the `PINT_CAP_BATCH_CREATE` bit set. This capability is currently granted only to servers. Clients may issue a *batch-create* request but they must provide a capability with the `PINT_CAP_CREATE` bit set and are only allowed to create a single object. The *handles* field of any capability is ignored.

6.2.3 Server `chdirent` state machine

The `chdirent` state machine modifies a directory entry to point to a new handle. Because the state machine modifies a directory it requires a capability with the `PINT_CAP_EXEC` and `PINT_CAP_WRITE` bits set. The capability should contain the handle of the directory whose entry will be modified.

6.2.4 Server `crdirent` state machine

The `crdirent` state machine creates a new directory entry that maps a name to a handle. Because the state machine modifies a directory it requires a capability with the `PINT_CAP_EXEC` and `PINT_CAP_WRITE` bits set. The capability should contain the handle of the parent directory.

6.2.5 Server `create` state machine

The purpose of the `create` state machine is to create new files in the file system. When a file is initially created in a PVFS file system it exists independently of a parent directory. In traditional UNIX^{TM} terms the file is similar to an inode with a reference count of zero. To support the creation of these initially orphaned files we introduce the `PINT_CAP_CREATE` capability, as described in . The `create` state machine will only permit the creation of new files if it receives a capability with the `PINT_CAP_CREATE` capability. The *handles* field of this capability is ignored.

In addition to creating the file in the file system the `create` state machine is also responsible for setting the file's initial attributes. These attributes are provided as part of the request sent by the client. The assignment of two of the file's attributes, the owner and group, is restricted by the `create` state machine. The state machine will not allow the owner of the file to be set to anything

other than the user id of the client. The file's group is restricted to one of the groups of which the client is a member. The user id and group membership information of the client is gathered from a credential that is sent as part of the file creation request. The file attribute checks are skipped if the capability has the `PINT_CAP_ADMIN` bit set.

6.2.6 Server `del-eattr` state machine

The purpose of the `del-eattr` state machine is to remove an extended attribute from a file's metadata. The state machine checks for a capability containing the target metadata handle and with the `PINT_CAP_SETATTR` bit set. The `PINT_CAP_SETATTR` capability is normally granted to the file's owner.

6.2.7 Server `get-attr` state machine

The purpose of the `get-attr` state machine is to retrieve the attributes for a PVFS object. Attributes include such items as the size and permissions of a file. The `get-attr` state machine ignores any capability because the attributes of an object are considered public knowledge.

The body of the `get-attr` state machine was modified to support the creation of capability objects. When a client needs a capability for a file system object it issues a request for the `PVFS_ATTR_CAPABILITY` attribute of that object. The request must include a credential object that authenticates the client to the server. The modified `get-attr` state machine generates and signs a capability for the client after taking into account any ACL entries for the object.

6.2.8 Server `io` state machine

The `io` state machine is the primary state machine controlling file I/O. It handles both reading from and writing to files. This state machine expects a capability whose *handles* field contains the file being read from or written to. If the file is being read the state machine requires the `PINT_CAP_READ` capability; if the file is being written the state machine requires the `PINT_CAP_WRITE` capability. .

6.2.9 Server `job-timer` state machine

The `job-timer` state machine resets certain performance counters on the server. This privileged operation requires a capability with the `PINT_CAP_ADMIN` bit set.

6.2.10 Server `lookup` state machine

The purpose of the `lookup` state machine is to map path names to handles. The incoming request contains a parent directory and the name of an entry to search for. The `lookup` state machine opens the directory data and returns the handle associated with the entry name if it exists. Because the state machine is searching the parent directory it requires a capability for the directory with the `PINT_CAP_EXEC` bit set.

The previous version of the `lookup` state machine contained an optimization that would allow it to lookup multiple path elements if they were all contained on the same server. However, because the capability implementation requires a separate capability for each path element this optimization was removed. As a result the client must perform two requests to search path elements: one to lookup the handle of the path element and the other to get a capability for that path element.

6.2.11 Server `mkdir` state machine

The purpose of the `mkdir` state machine is to create new directories in the file system. When a directory is initially created in a PVFS file system it exists independently of a parent directory. In traditional UNIX™ terms it is similar to an inode with a reference count of zero. The ability of a directory to exist without a parent directory adds a certain degree of complexity to the code responsible for authorizing directory creation requests.

6.2.12 Server `prelude` state machine

The `prelude` state machine runs before each of the other server state machines begins processing a request. It helps avoid uneccessary duplication of code by performing operations that are common to every state machine. Among its responsibilities are adding the request to the job scheduler, reading the basic attributes of any target handle, translating user and group ids, verifying

credentials and capabilities, and executing the correct permission check routine for the request. As a nested state machine it does not perform any permission checks itself.

6.2.13 Server `readdir` state machine

The purpose of the `readdir` state machine is to list the entries in a directory. The state machine requires a capability with the `PINT_CAP_READ` bit set for the directory being read.

6.2.14 Server `rmdirent` state machine

The `rmdirent` state machine removes an entry from a directory. It does not remove the object to which the entry points. Because the state machine modifies a directory it requires a capability with the `PINT_CAP_EXEC` and `PINT_CAP_WRITE` bits set. The capability should contain the handle of the parent directory.

6.2.15 Server `set-attr` state machine

The `set-attr` state machine is responsible for setting the attributes of a file system object, typically a file or directory. The state machine requires a capability for the target object with its `PINT_CAP_SETATTR` bit set. This capability is normally granted for the owner of the file.

Extra permission checks are performed inside the body of the state machine to protect certain sensitive attributes. Among these attributes are the owner and group of an object. The state machine does not allow a user without the `PINT_CAP_ADMIN` capability to set an object's owner to anything other than the user id of the client. Additionally, the object's group must be one of the groups of which the client is a member. User and group information for the client is provided by a credential object sent with the request.

6.2.16 Server `set-eattr` state machine

The purpose of the `set-eattr` state machine is add an extended attribute to a file's metadata. The state machine checks for a capability containing the target metadata handle and with the `PINT_CAP_SETATTR` bit set. The `PINT_CAP_SETATTR` capability is normally granted to the file's owner.

6.2.17 Server `small-io` state machine

The `small-io` state machine supports reading from and writing to small files. The access check semantics of the `small-io` state machine are identical to those of the `io` state machine as described in Section 6.2.8.

6.2.18 Server `truncate` state machine

The purpose of the `truncate` state machine is to change the size of a file. The state machine checks for a capability with the `PINT_CAP_WRITE` bit set and containing the target file.

Bibliography

[1] 1003.1 standard for information technology—portable operating system interface (POSIX) base definitions, issue 6. *IEEE Std 1003.1-2001. Base Definitions, Issue 6*, pages i–448, 2001.

[2] 1003.1 standard for information technology—portable operating system interface (POSIX) system interfaces, issue 6. *IEEE Std 1003.1-2001. System Interfaces, Issue 6*, pages i–1690, 2001.

[3] Philip H. Carns. *Achieving scalability in parallel file systems.* PhD thesis, Clemson University, Clemson, SC, USA, 2005.

[4] Cray, Inc. Cray Inc. announces 2008 acceptance of the "petaflops" supercomputer at Oak Ridge National Laboratory, April 2011. `http://investors.cray.com/phoenix.zhtml?c=98390&p=irol-newsArticle&ID=1240189`.

[5] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *Proceedings of the ClusterWorld Conference and Expo, in conjunction with the 4th International Conference on Linux Clusters: The HPC Revolution*, 2003.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008.

[7] Jack B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM*, 12:589–602, October 1965.

[8] V.D. Gligor and B.G. Lindsay. Object migration and authentication. *Software Engineering, IEEE Transactions on*, SE-5(6):607–611, November 1979.

[9] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem.* PhD thesis, Carnegie Mellon University, July 1999.

[10] R.Y. Kain and C.E. Landwehr. On access checking in capability-based systems. *Software Engineering, IEEE Transactions on*, SE-13(2):202–207, February 1987.

[11] Khronos Group. OpenCL, April 2011. `http://www.khronos.org/opencl/`.

[12] B. W Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, pages 437–443. Princeton University, March 1971.

[13] Henry M. Levy. *Capability-based computer systems.* Digital Press, Bedford, Mass., 1984.

[14] H.W. Meuer, E. Strohmaier, and J.J. Dongarra. TOP500 supercomputer sites, 36th edition. In *ACM/IEEE Proceedings of the Supercomputing Conference (SC2010)*, November 2010.

[15] NVIDIA Corporation. What is CUDA?, April 2011. `http://www.nvidia.com/object/what_is_cuda_new.html`.

[16] S.A. Rajunas, N. Hardy, A.C. Bomberger, W.S. Frantz, and C.R. Landau. Security in KeyKOS. *Security and Privacy, IEEE Symposium on*, 0:78, 1986.

[17] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, February 1978.

[18] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *USENIX Conference Proceedings*, pages 119–130, Portland, OR, Summer 1985. Sun Microsystems, Inc., USENIX.

[19] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, Sun Microsystems, Inc., C. Beame, Hummingbird Ltd., M. Eisler, D. Noveck, and Network Appliance, Inc. Network File System (NFS) version 4 protocol. RFC 3530, April 2003.

[20] Sun Microsystems. System and network administration. March 1990.

[21] Inc. Sun Microsystems. NFS: Network File System protocol specification. RFC 1094, March 1989.

[22] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 558–563, 1986.

[23] F. Wang, Q. Xin, B. Hong, S.A. Brandt, E.L. Miller, and D.D.E. Long. File system workload analysis for large scientific computing applications. In *Mass Storage Systems and Technologies, 2004. Proceedings. 21st IEEE / 12th NASA Goddard Conference on*, April 2004.

[24] Simon Wiseman. A secure capability computer system. *Security and Privacy, IEEE Symposium on*, 0:86, 1986.