UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

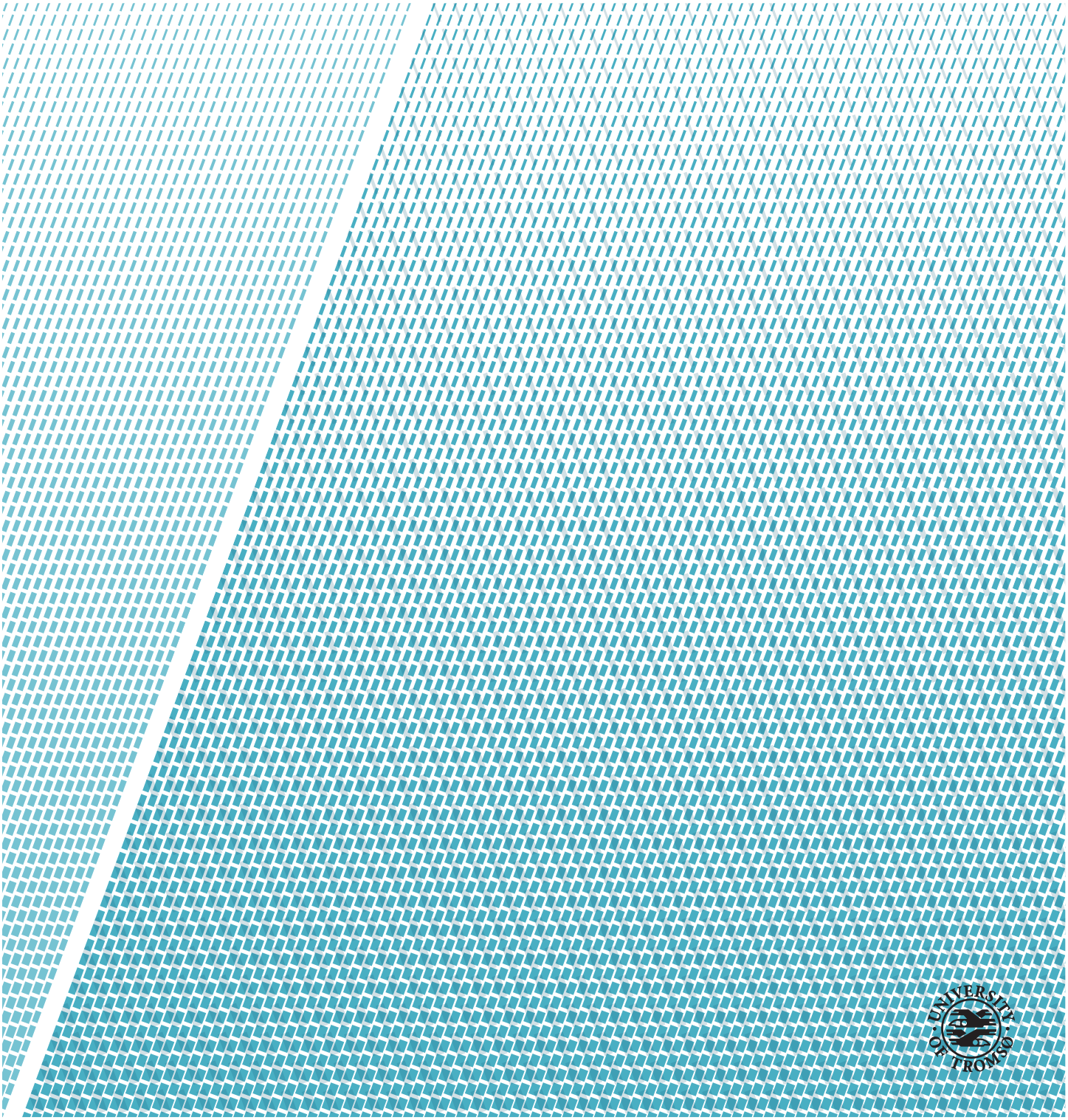
Faculty of Science and Technology
Department of Computer Science

Implementing an SMB2 Server in the Vortex Operating System

—

Vegard Sandengen

INF 3990 — Master's thesis in Computer Science, May 2015



“I stedet for å bli sint på tingene, så lar jeg nå heller tingene få se hva som skjer hvis de får det som de vil.”
–Atle Antonsen
(Tre brødre som ikke er brødre, 2005)

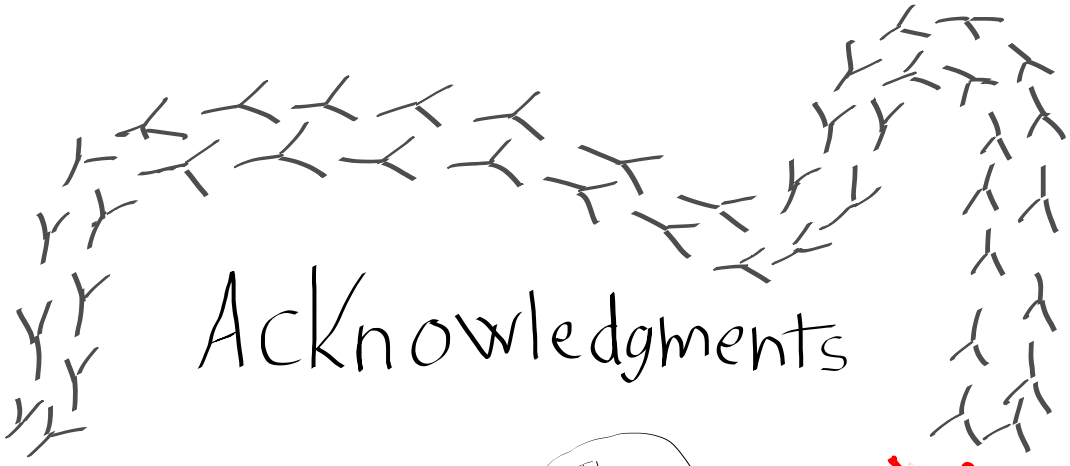
Abstract

With the advent of computer networks, the ability for sharing and accessing files across the network between multiple workstations and remote servers was sought after. In the nineteen eighties, prominent networked file systems were developed and reached widespread adoption among enterprise businesses and institutions. A few of these, notably Networked File System (NFS) and Server Message Block (SMB), survived the transition into the Internet era and the successors of these protocols remain the default network file systems on contemporary operating systems today.

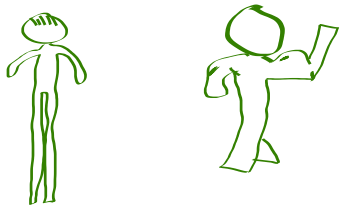
Clouds are comprised of thousands of computers, hosted in centralized data center facilities. These computers run modified versions of contemporary operating systems, with a monolithic, micro or hybrid kernel. Contemporary operating systems lack fine-grained control over resource allocation. The Omni-kernel architecture is a novel operating system architecture designed for pervasive monitoring and scheduling of system resources. Vortex is an experimental implementation of the Omni-kernel architecture. The Vortex operating system lack utilities to expose its native file system over the network.

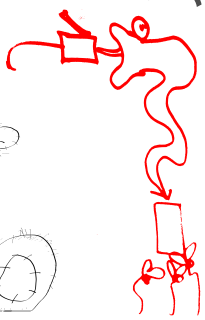
This thesis describes the introduction of a minimal Server Message Block version 2 (SMB2) server to the Vortex operating system. We achieve interoperability with contemporary client(s) and document acceptable throughput performance.

Acknowledgments



My advisors





Contents

List of Figures

List of Structures

List of Abbreviations

ABI	application binary interface
ADT	Abstract Data Type
AFS	Andrew File System
API	application programming interface
ASCII	American Standard Code for Information Interchange
CAT	Common Authentication Technology
CIFS	Common Internet File System
CPU	Central Processing Unit
DCE/RPC	Distributed Computing Environment / Remote Procedure Calls
DES	Data Encryption Standard
ELF	Executable and Linkable Format
FUSE	file system in userspace
GSS	Generic Security Service
GSS-API	Generic Security Service - Application Programming Interface
HDFS	Hadoop Distributed File System

I/O	Input / Output
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPC	interprocess communication
ISA	instruction set architecture
KDC	key distribution center
LAN	Local Area Network
LLC	Logical Link Control
NBF	NetBIOS Frames protocol
NBT	NetBIOS over TCP/IP
NBX	NetBIOS over IPX/SPX
NCP	NetWare Core Protocol
NetBEUI	NetBIOS Extended User Interface
NetBIOS	Network Basic Input/Output System
NFS	Networked File System
NIC	Network Interface Card
OID	object identifier
OS	operating system

OSI	Open System Interconnection
PDU	protocol data unit
RDMA	Remote Direct Memory Access
RFC	Request For Comment
RFS	Remote File Sharing
RoCE	RDMA over Converged Ethernet
RPC	Remote Procedure Call
SLA	service-level agreement
SLO	service-level objective
SMB	Server Message Block
SMB1	Server Message Block version 1
SMB2	Server Message Block version 2
SPNEGO	Simple and Protected Generic Security Service Negotiation Mechanism
TCP	Transmission Control Protocol
VFS	virtual file system
VM	virtual machine
VMM	virtual machine monitor
WAN	Wide Area Network

1 Introduction

Over the last decade, cloud computing and storage services have revolutionized small-scale web businesses and how society interacts with the Internet. A cloud provider offers distributed, geo-replicated infrastructure with low-cost entry, where customers only pay for the resources they consume. Expensive start-up and upgrade costs are a thing of the past.

Clouds are served from thousands of computers, or nodes, hosted in centralized data center facilities. These nodes run modified versions of contemporary operating systems, with a monolithic, micro, or hybrid kernel. Contemporary operating systems lack fine-grained control over resource allocation, typically resulting in the cloud provider over-provisioning tenant computing resources so as to reduce the risk of service-level agreements (SLAs) or service-level objectives (SLOs) [1] violation. The omni-kernel architecture is a novel operating system (OS) architecture designed for pervasive monitoring and scheduling of system resources [2] [3]. The control over resource allocation permitted by the omni-kernel may eliminate much of the over-provisioning required when employing other kernel architectures.

Vortex is an experimental implementation of the omni-kernel architecture providing a novel light-weight approach to virtualization [2] [3] [4] [5] [6] [7]. Paravirtualization technology in conventional virtual machine monitors (VMMs) offers virtualized device interfaces to guest virtual machines (VMs) [8], while Vortex is capable of offering high-level commodity OS abstractions. It aims to offload common OS functionality from its VM OSs, reducing both the resource

footprint of a single VM and the duplication of functionality across all guest OSs. Because of this approach to paravirtualization, Vortex does not provide a complete virtualization environment for ports of commodity OSs; but rather, a light-weight emulation approach supporting the application binary interface (ABI) and system call interface of the selected OS. A VM OS capable of running Linux applications such as Apache[1], MySQL[2] and Hadoop[3] exists for Vortex [5] [6].

Cloud environments offer storage as a service infrastructure, facilitating distributed and replicated data in different failure domains. The storage infrastructure is implemented through specialized file systems, such as Google FS [9], Microsoft Azure [10], Lustre[4] and Hadoop Distributed File System (HDFS).[3] These do not offer a conventional file system view, but rather a blob storage in separated namespaces. Given a partitioned set of nodes with specialized or administrative responsibilities, they may require local file systems to be exposed over the network, if only within the data center itself. Conventional networked file systems deployed in contemporary operating systems accomplish this, with out-of-the-box interoperability from a remote work station.

Many of the prominent proposals and implementations of distributed and networked file systems arose in the nineteen eighties. Andrew File System (AFS) developed by Carnegie Mellon University as part of an ambitious research project. AT&T's Remote File Sharing (RFS), Networked File System (NFS) version 2 by Sun Microsystems, Novell's NetWare Core Protocol (NCP), and Server Message Block (SMB) by several organizations, among others. Decades later, successors of the NFS and SMB protocols are still the default network file systems on contemporary operating systems.

The Vortex operating system lack utilities to expose its native file system over the network. NFS relies on a set of Remote Procedure Call (RPC) mechanisms, requiring a complete RPC framework to complete. SMB on the other hand, exchanges a set of protocol data units (PDUs) on top of a reliable network transport, such as Transmission Control Protocol (TCP). In that same regard, SMB has been the default network file system in Microsoft operating systems going all the way back to the early nineties. Vortex needs a networked file system to become feature complete, and SMB is an obvious choice to fulfill this requirement.

1. http://httpd.apache.org
2. http://www.mysql.com
3. http://hadoop.apache.org
4. http://www.lustre.org

1.1 Thesis statement

We will enhance Vortex with a native application implementing an Server Message Block version 2 (SMB2) dialect 2.002 server, exposing its native file system.

> *This thesis shall cover design, implementation, deployment and evaluation a versatile, minimalistic SMB2 server for the Vortex operating system. This should allow Vortex to expose the native file system and be compatible with a commodity SMB client.*

The thesis will not concern itself with exposing the services offered through interprocess communication (IPC) mechanisms. The implementation need not support every permutation of exotic and rarely used commands defined by the protocol, but should still offer the basic operations expected from browsing a file system from a client on a conventional desktop through its native file browser.

1.2 Methodology

This thesis follows a system development methodology. We seek to construct a system for solving a given problem. The requirements and restrictions of the problem is laid forth prior to developing a prototype. The system is continuously tested during development to determine whether or not it solves the problem at hand. If the prototype fulfills the requirements, the system is subject to performance measurements and final evaluation in line with the problem statement.

1.3 Outline

The remainder of this thesis is structured as follows:

Chapter 2 introduces the history of a legacy transport mechanism, NetBIOS, employed for Server Message Block version 1 (SMB1) which still affects the recent versions. The Executable and Linkable Format (ELF) file format is explained, and its role made clear in Chapter 4. Finally, the authentication mechanisms used to establish a secure session in SMB2 is described.

Chapter 3 covers the technical details of the SMB2 protocol format. It clarifies inconsistencies in the technical document [11] and attempts to describe the details behind key mechanisms.

Chapter 4 describes the server architecture, presenting its versatile design and benefits thereof. Key implementation difficulties are highlighted and their solutions laid forth.

Chapter 5 evaluates our implementation This is done with experiments that exercise key SMB features and through comparisons to other implementations running on commodity OSs.

Chapter 6 discusses future work and areas of improvement, and concludes.

2 Background

SMB is a client-server protocol, capable of providing all the functionality offered by a local file system. The server exposes a portion of its file system to authenticated clients, who issue network commands that map to operations against the servers local file system. This enables a desktop computer, an SMB client, to access remote files and directories through its own file system view, as if they were stored locally.

This chapter describes the history, functionality, and concepts relevant to the background of SMB and this thesis. Section 2.1 explains the history of the NetBIOS transport protocol, essential for complete understanding of legacy constraints accompanying SMB today. Section 2.2 introduces the ELF file format, required to explain implementation difficulties when adapting external libraries written for Unix to an emulated environment in Vortex. Section 2.3 describes a generic authentication mechanism employed in SMB.

2.1 NetBIOS

NetBIOS is a network transport used for the original versions of SMB. It still has some relevance to the latest versions, as highlighted in this section.

In the mid eighties IBM announced its first Local Area Network (LAN) system, complete with both networking hardware and software abstractions to

communicate among the connected devices, called IBM PC Network [12]. The LAN infrastructure operated in two different modes: broadband or baseband. Both modes were bus-attached LAN that could accommodate up to 72 and 80 connected devices, respectively.

A new software application programming interface (API), named Network Basic Input/Output System (NetBIOS), supported their new LAN environment. Over PC Network, the supplied implementation of NetBIOS relied on proprietary networking protocols to communicate over the wire. The initial intent behind NetBIOS was to be an interface between an application and the networking adapter, abstracting the hardware specific code away from the application source code.

The services available through the NetBIOS API are as follows:

Name Service to associate each application with an unique 16 octet NetBIOS name. This service is essential to start operating the other services provided, since they reference each endpoint using the NetBIOS name.

Datagram Service to distribute connectionless packets, either single targeted or broadcasted. The application is responsible for error detection and recovery, and delivery is not guaranteed.

Session Service provides a connection oriented environment where messages can span multiple packets. Error detection and recovery of lost packets is the responsibility of the session service.

NetBIOS belongs in the session layer of the OSI Reference Model [13]. Figure 2.1 shows an illustration of the OSI Reference Model. It is a protocol solely designed for the PC Network, with its maximum capacity of 80 connected devices. For instance, the original protocol resolved NetBIOS names through a distributed broadcast, expecting the node with the requested name to respond. If the node issuing the name query do not receive a response within a reasonable time frame — in the multiple seconds scale — the NetBIOS name queried is considered available and not in use by any other node on the network. Since the resolution method involved a broadcast to each connected node, the amount of traffic on the network and packets processed by each node scales linearly with the interconnected node size. Thus, the effectiveness of the network as whole decreases equally.

In 1985, IBM launched its Token-Ring LAN scheme to replace the PC Network, increasing their node capacity to roughly 270 connected devices. Since the original proprietary NetBIOS implementation only suited the PC Network infrastructure, IBM required a new implementation to support their Token-Ring

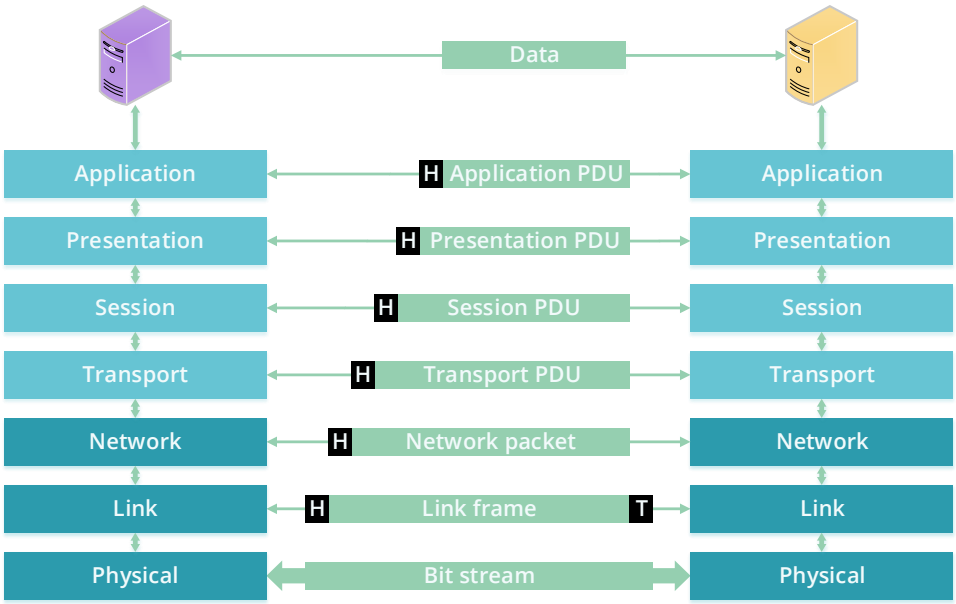


Figure 2.1: An illustration of the OSI Reference Model. Modern interpretation and protocol suites compress the session and presentation layer into the application layer. The lower three layers, Network, Link and Physical represented with deep contrasted rectangles, are packets navigating through the network, processed by individual routers and switches along the way. The upper four layers, represented by dimly colored boxes, are only processed by the sending and receiving computers on the network.

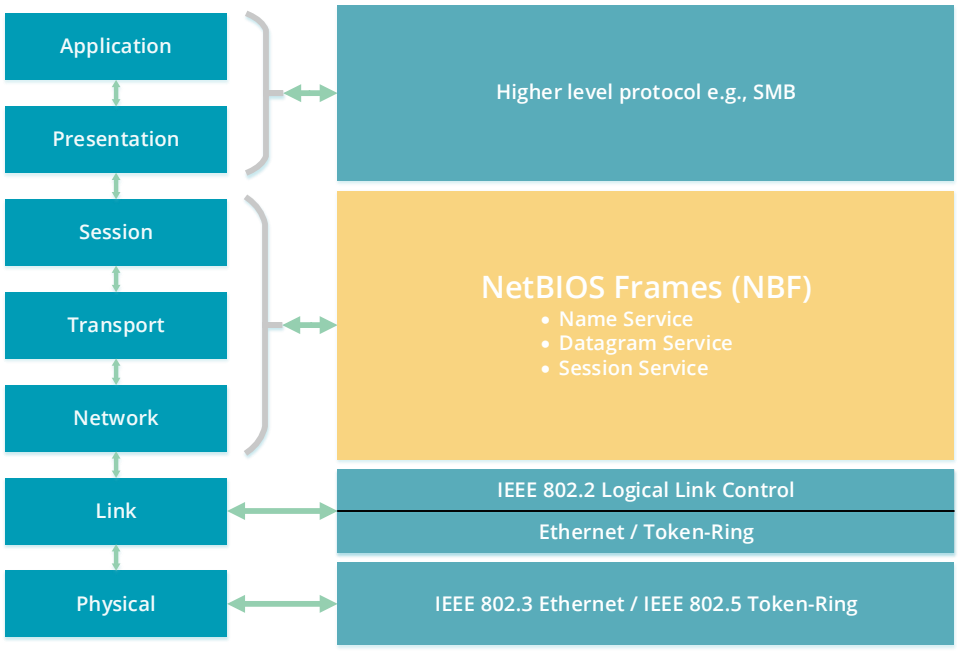


Figure 2.2: An illustration of how NBF positions itself in the OSI Reference Model.

LAN network. This new implementation was called NBF and was developed in conjunction with an extension to the NetBIOS API, namely NetBIOS Extended User Interface (NetBEUI) [14]. The NBF protocol is often confused with NetBEUI due to Microsoft erroneously labelling its NBF protocol implementation NetBEUI. NBF is simply a protocol implementing the API specified by NetBEUI. For the sake of backwards compatibility with NetBIOS aware applications designed for PC Network, a NetBEUI *emulator* enabled these applications to continue functioning on the Token-Ring LAN. The services provided by Token-Ring implements Logical Link Control (LLC), standardized in IEEE 802.2 [15], which is the upper portion of the link layer in theOSI Reference Model. Token-Ring is a standard protocol specified in IEEE 802.5 [16].

The NBF protocol resides in the network layer, relying on the IEEE 802.2 LLC for data transmission. This implies that NBF may function equally well, without modification, on a different link layer and is not tied to Token-Ring, even though their development coincided. NBF thus runs on Ethernet as well. Figure 2.2 illustrates the position of NBF and Token-Ring in relation to other networking interfaces on the OSI Reference Model.

There exists no standard or formal specification for any of the protocol(s) used with NetBIOS; in practice only the technical reference documents from IBM serve this purpose [14]. Several implementations of the NetBIOS protocols were reverse-engineered from the original, containing small discrep-

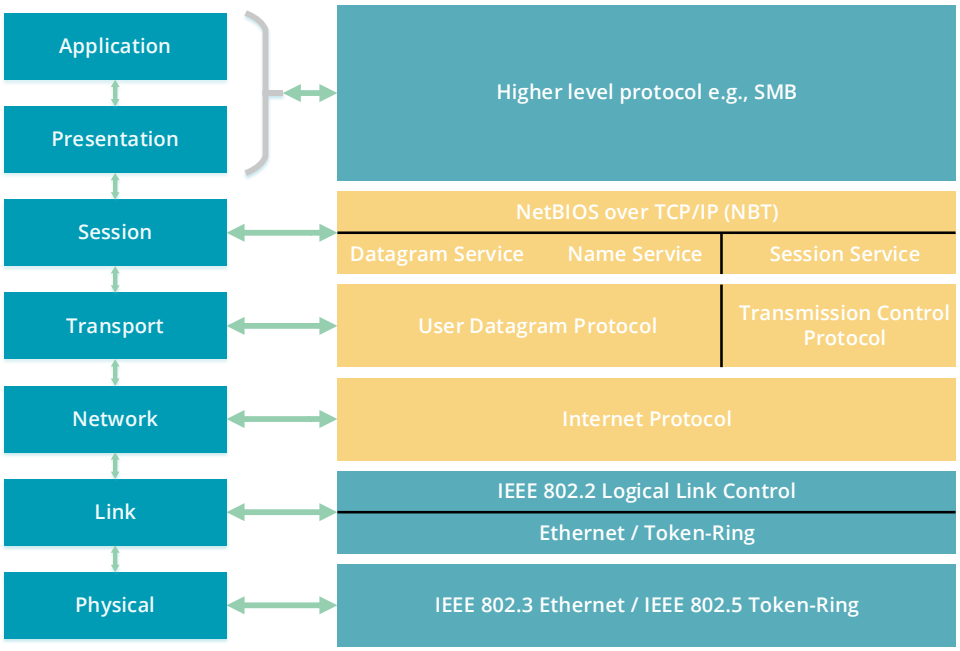


Figure 2.3: an illustration of how NBT positions itself in the OSI Reference Model in relation to TCP/Internet Protocol (IP) and the physical/link layer.

ancies making them generally incompatible with each other. This lack of standardization makes understanding the underlying networking primitives difficult [17].

The invention of the Internet required a change of protocols in NetBIOS to enable its applications to inter-operate on the modern TCP/IP network. This change brought the standardization of NetBIOS over TCP/IP (NBT) [18][19], with better network scaling properties. The name resolution broadcast of NBF was replaced with a centralized server in charge of name resolution. Each machine/application requires a configured server it can contact to perform both registration, deletion, and resolution. Only machines with the same dedicated naming server can inter-operate in NBT. The centralized server effectively reduced the processing time of the name resolution service, since the communication happens directly with one single entity and not a broadcast to every participating node on the network. Figure 2.3 illustrates the relationship between NBT, OSI Reference Model, and the Internet protocol suite.

2.2 ELF - Object format

The ELF object format is relevant for our implementation of the GSS-API authentication subsystem, described in Section 4.3. ELF is the de facto file format on Unix-like systems, used for both object files, shared libraries, and executables. It was designed by AT&T Unix System Laboratories and first specified in the System V release 4 ABI [20]. Tools Interface Standards, a separate organization, later extracted the definition of ELF, embedded in the System V ABI specification, and published it as a separate standard [21].

ELF is a versatile file formate that is extensible by design, and does not assume any particular processor or architecture. It's structured with a single well-defined ELF header, identified by the first four bytes containing the sequence \x7F followed by 'ELF' in ASCII encoding. The remainder of the file is data, formatted in one of three ways:

Section header table consists of fixed-size entries each describing a continuous range of non-overlapping bytes, called a section. There may only be one Section header table in an ELF formated file, whose offset, length and entry size information is specified in the ELF header. Each entry describing a section contains information relating to its type, memory requirements, alignment constraints, linkage, flags, and additional type-dependent info.

Sections are a non-unique, named continuous ranges of arbitrary data. The name of a section defines its purpose. Sections hold the bulk of object file information for the linking view, such as instruction, symbol table, relocation information, and initialized data.

Program header table describes how the system should create a process image. It consists of fixed-sized entries describing a segment, spanning one or more sections. Each segment describes how its range of sectors should be represented in memory.

Furthermore, the ELF header identifies architecture bit length, endianness, target operating system ABI, targeted instruction set architecture (ISA), and the description of both section and program table header.

An ELF file may hold any set of arbitrary data; however, two use cases stem from its conception. Object files require the section table header, with multiple different section types, not just the data and instruction ones used for executables. Segments are not defined for an object file, since segment layout in memory requires relocation and linking dependencies to be determined. The Program header table is thus only meaningful to executables and shared object

Section name	Description
.BSS	Holds uninitialized data contributing to the program's memory image. This section occupies no file space; rather, it is zero-initialized when the file is unpacked for execution by the operating system.
.DATA .DATA1	Initialized data that is a part of the program's memory image.
.RODATA .RODATA1	Section contains read-only data part of a non-writable segment in the process image.
.DYNAMIC	Contains dynamic linking information.
.DEBUG	Implementation-defined debugging information.
.SHSTRTAB	Holds ASCII encoded string names for all sections.
.STRTAB	Section contains strings, referenced within the ELF file itself. Most commonly it holds name references from the .SYMTAB.
.SYMTAB	Holds a well-formated Symbol Table describing symbols referenced within the object file. These entries will have offset and length variables referencing their name from the .STRTAB section.
.TEXT	Contains the executable instruction data of a program.

Table 2.1: Table of special ELF sections and descriptions of their uses.

files; which in turn do not rely on the section header table, which may be safely stripped from these file types.

Table 2.1 lists and describes the most common reserved section types. The only required sections for an executable, described by one or multiple segments, are the data and text sections. The .DEBUG section, and derivatives of this, present debugging information from any high-level language to debuggers, supporting arbitrary platforms and ABIs.

The ELF file format is important to the implementation and facilitation of the Generic Security Service - Application Programming Interface (GSS-API) authentication system, described in Subsection 4.3.2

2.3 GSS-API

The GSS-API is an interface to provide authentication services between a client and server. It is the only method used to establish a secure session in SMB2. Without this mechanism in place, one cannot create a compliant SMB2 server

nor client.

In 1991 the Internet Engineering Task Force (IETF) assembled a working group named Common Authentication Technology (CAT). Its goal was to provide standardized distributed security services in a manner which insulated the applications utilizing the services from the underlying security mechanisms [22]. This relieved application developers from embedding and handling security-specific implementation tasks in their protocols, and rather facilitated a transport of opaque tokens passed to and from the generic security service. This also leaved the implementation of critical security infrastructure to an expert group, instead of each protocol implementing or adapting their own.

CAT submitted the first version of their work two years later, naming their product Generic Security Service - Application Programming Interface (GSS-API) [23]. It defined a standard interface and accompanying bindings for the C programing language [24][25]. The latest released version of GSS-API is version 2, update 1 standardized at the turn of the millennium [26][27].

The GSS-API defines 45 procedure calls to offer its services, which facilitates:

Authentication to confirm the identity of both communicating parties.

Integrity of the exchanged protocol data, avoiding tampering by a man-in-the-middle attack.

Confidentiality such that only the two parties are able to view the contents of their communication.

Authorization is the ability to determine access rights to a set of resources. GSS-API only provides authorization services through implementation-defined mechanisms. One caveat of Generic Security Service (GSS) is that it assumes a client-server architecture. Listed in Code Snippet 2.1 are the most prominent methods to handle authentication. Code Snippet 2.2 depicts pseudo-code of how the server uses the authentication API.

The GSS-API library, which applications interface against, does not implement any of the security services; rather, it facilitates third-party libraries to implement the GSS-API interface as a *mechanism*. These will in turn perform the required services in an implementation-defined manner. Applications thus link against the facilitating GSS-API library, not directly against the implementing interface. This is illustrated in Figure 2.4.

All mechanisms are identified by a well-known object identifier (OID), which is

Code Snippet 2.1 : C Prototype definitions of the most significant GSS-API functions who provide authentication services

```
// Convert username/hostname into a form that identifies a security entity
OM_uint32 gss_import_name(          OM_uint32                       *minor_status,
                                    const gss_buffer_t              input_name_buffer,
                                    const gss_OID                   input_name_type,
                                    gss_name_t                      *output_name
);

// Obtain identity
OM_uint32 gss_acquire_cred(         OM_uint32                       *minor_status,
                                    const gss_name_t                desired_name,
                                    OM_uint32                       time_req,
                                    const gss_OID_set               desired_mechs,
                                    gss_cred_usage_t                cred_usage,
                                    gss_cred_id_t                   output_cred_handle,
                                    gss_OID_set                     *actual_mechs,
                                    OM_uint32                       *time_rec
);

// Client generates token to server.
OM_uint32 gss_init_sec_context(     OM_uint32                        *minor_status,
                                    const gss_cred_id_t              initiator_cred_handle,
                                    gss_ctx_id_t                     *context_handle,
                                    const gss_name_t                 target_name,
                                    const gss_OID,                   mech_type,
                                    OM_uint32                        req_flags,
                                    OM_uint32                        time_req,
                                    const gss_channel_bindings_t     input_chan_bindings,
                                    const gss_buffer_t               input_token,
                                    gss_OID                          *actual_mech_type
                                    gss_buffer_t                     output_token,
                                    OM_uint32                        *ret_flags,
                                    OM_uint32                        *time_rec
);

// Processes token (generated from gss_init_sec_context) from client.
// Produces a response token to return.
OM_uint32 gss_accept_sec_context(   OM_uint32                       *minor_status,
                                    gss_ctx_id_t                    *context_handle,
                                    const gss_cred_id_t             acceptor_cred_handle,
                                    const gss_buffer_t              input_token_buffer,
                                    const gss_channel_bindings_t    input_chan_bindings,
                                    gss_name_t                      *src_name,
                                    gss_OID                         *mech_type,
                                    gss_buffer_t                    output_token,
                                    OM_uint32                       *ret_flags,
                                    OM_uint32                       *time_rec
                                    gss_cred_id_t                   *delegated_cred_handle
);
```

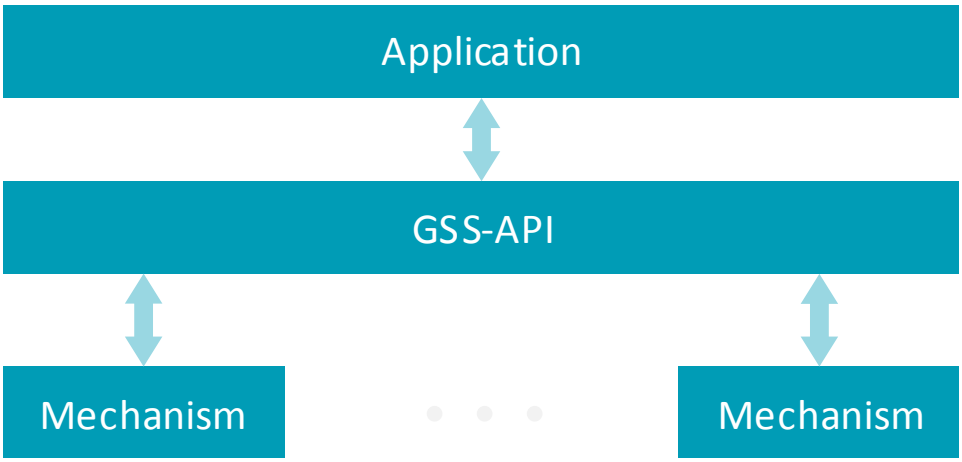


Figure 2.4: This figure illustrates how the GSS-API is architecturally situated between the application and the underlying set of mechanisms that implement the security services.

a hierarchically-assigned namespace where each node is managed by separate authorities. The client initiates communication with its preferred mechanism, either explicitly specified in the application or resolved in a system-dependent fashion in the GSS-API implementation. The OID of the client-selected mechanism is encoded in the token. When the server processes this token, it performs a lookup in its set of mechanisms to identify a match. If the server cannot find a match, it returns an error code and leaves the responsibility to handle the failed security context establishment to the application protocol.

This static fashion of selecting a mechanism would possibly require a recompilation of the application if a "switch of mechanism" was needed. There is no method of negotiating supported mechanisms by both parties in GSS itself; however, CAT addressed this with the development of a pseudo-mechanism dubbed Simple and Protected Generic Security Service Negotiation Mechanism (SPNEGO) [28][29]. When used as the selected mechanism, it exchanges the set of supported mechanisms between the client and server. The first intersecting mechanism of the two exchanged sets is selected and SPNEGO henceforth directs all GSS procedure calls to the negotiated mechanism. Figure 2.5 depicts this relationship.

SMB2 requires use of SPNEGO in conjunction with authentication over GSS-API.

Code Snippet 2.2 : An example usage of the authentication procedures of the GSS-API to establish a security context between a newly accepted client. CLIENT_AUTHENTICATE is invoked on a separate thread and will block on CLIENT_RECEIVE_* until an opaque token sent through the application protocol is received from the client. The token is sent through the GSS-API and produces a response token returned back to the client through the application protocol. Repeat until the security context is established.

```
void server_acquire_credentials(gss_cred_id_t *server_credentials)
{

    // Acquire the servers credentials.
    // Parameters may be passed as 'no specified', such that system defaults are used.
    // May acquire credentials for all mechanisms available on the system.
    major_gss_status = gss_acquire_cred(
                                        :
                                        :

                                        &server_credentials,
                                        :
                                        :

                                        );
    if (major_gss_status != GSS_S_COMPLETE)
    // error

    return;
}
// Function is invoked when a new client is accepted on the connection.
void client_authenticate(gss_ctx_id_t *client_context, gss_cred_id_t *client_credentials)
{
    OM_uint32           major_gss_status;
    gss_buffer_desc     input_buffer;
    gss_buffer_desc     output_buffer;
    gss_cred_id_t       server_credentials;

    server_acquire_credentials(&server_credentials);

    while (1)
    {
        // Get new opaque token from the client transferred through the application protocol.
        // Blocking operation until the another packet is received from the client and
        // the token is extracted from the protocol.
        // Input buffer is populated with this opaque token.
        client_receive_gss_token(&client_context, &input_buffer);

        // Send the token down for processing. If we do not implement the received mech,
        // major_gss_status will be GSS_S_BAD_MECH.
        major_gss_status = gss_accept_sec_context(  ...
                                                    client_context,
                                                    server_credentials,
                                                    &input_buffer,
                                                    :
                                                    :

                                                    &output_buffer,
                                                    ...
                                                    client_credentials
                                                    );
        if (major_gss_status != GSS_S_COMPLETE && major_gss_status != GSS_S_CONTINUE_NEEDED)
        // Error - cannot establish security context

        // Send the opaque token response back to client though the application protocol.
        client_send_gss_token(&output_buffer);

        if (major_gss_status == GSS_S_COMPLETE)
        {
            // finished - Security context established.
            break;
        }

        // More processing required - Go another round to get new token from client.
    }
    return;
}
```

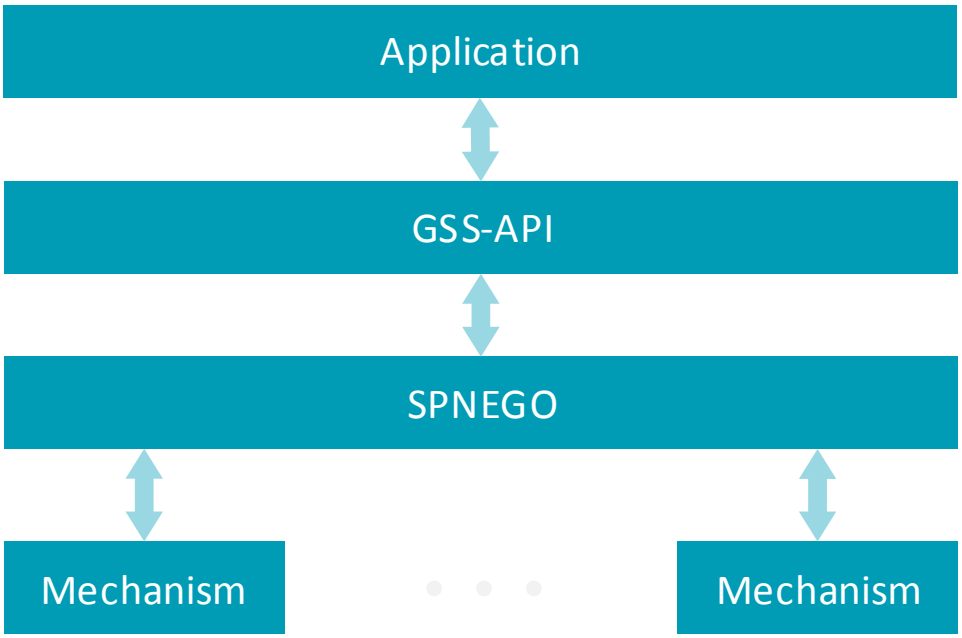


Figure 2.5: Illustration of how SPNEGO is used in GSS to provide negotiating facilities for available mechanisms in a system.

2.3.1 Protocol details

The GSS-API contains two methods to establish an authenticated context between two communicating parties. The initiator, usually the client, uses the `gss_init_sec_context()` function to both generate and further establish the session. The server uses the output supplied by the initiator as input into the server side `gss_accept_sec_context()`, and replies with its output token which in turn is processed by the client. The initiator requires a specific mechanism supplied to the method, whilst the acceptor determines if it supports the clients mechanism through its credential handle. Both may specify to use system defaults. The details of transferring the selected mechanism, as well as the encoded data is the topic of this subsection.

The GSS protocol defines two crucial constructs used to communicate and relay the encapsulated data to the underlying mechanism. These are encoded in ASN.1 syntax, so that endianess and data type sizes are serialized between communicating parties.

The first construct is *MechType*, which is a self-describing variable number of octets encoding an OID. The first byte is an ASN.1 tag for OID, with the value 0x06. Following this, a variable number of octets are used to encode the total length of both the self-describing value field and the subsequent OID field. The

length is encoded in one of two ways: (1) if the value is less than 128 bytes, a single octet is used with the high order bit set to zero, and the 7 low order bits encode the value, or (2) if the value is 128 bytes or larger, with the first octet having its high order bit set to one, and the remaining 7 low order bits encode the number of subsequent octets, 8 bits per octet, carrying the value with the most significant digit first.

The second construct is an *InitialContextToken*, which encapsulates both the *MechType* and the subsequent mechanism-specific blob. An illustration is available in Figure 2.6. It is constructed with the first byte being a well-defined ASN.1 tag for application-defined encoding, indicated by the value 0x60. It is followed by a variable number of octets describing the total length of the remaining blob, including the octets used to encode the length. This self-describing value is encoded across multiple octets in the same way as described above. The next expected sequence is the *MechType* construct, encoding the OID of the selected mechanism. The remaining data, known as the *innerContextToken* is mechanism-specific and may be encoded in any way. It derives its length from the value encoded in the token length field following the application tag. Another common name for this *innerContextToken* is *mechToken*.

All initial calls to `gss_accept_sec_context()` on a given mechanism must — according to the RFCs — contain an *InitialContextToken* with the OID of the selected mechanism to establish a new context. This is required such that the acceptor may deduce which mechanism the client is communicating on. Subsequent invocations with an established context expect only mechanism-specific data, and has no restriction on encoding.

SPNEGO

SPNEGO uses the *innerContextToken* to encode an exchange of supported mechanisms, and it allows the client to supply an opportunistic mechToken of its preferred mechanism encapsulated within SPNEGO *innerContextToken*. If the initiators preferred mechanism is chosen and it supplied an opportunistic mechToken for this mechanism, this must in turn be relayed to `gss_accept_sec_context()` with an empty context.

This is a point of confusion since the acceptor function is invoked a second time with a different mechanism, and by extension a new, empty context. This implies that the opportunistic mechToken must in turn be formated as an *InitialContextToken*, containing its preferred mechanism OID in the *MechType* field as well as an *innerContextToken*.

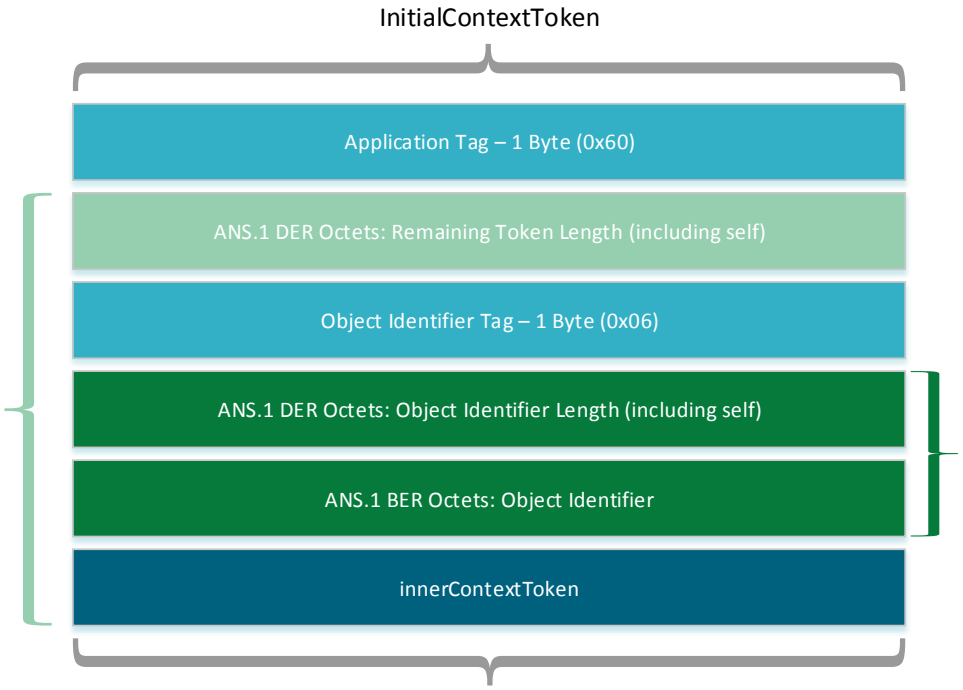


Figure 2.6: Illustration of the components making up the InitialContextToken of GSS-API which is the expected format when establishing new contexts through a mechanism of GSS. The application tag byte is a required ASN.1 encoding by signifying that all following bytes are application specific. Whereupon the total length of the remaining blob is encoded, which includes the octets used to encode the length. Another ASN.1 Tag is present indicating that an OID is encoded following this byte. The length of this OID is encoded before the OID itself follows. The remaining bytes, deduced from value of the Remaining Token Length field, are mechanism-specific and is referenced in the specification as innerContextToken.

2.3.2 Unix implementation

There exists several implementations of the GSS-API and associated mechanisms. The Kerberos version 5 adaption to GSS was co-produced alongside the specification itself by CAT. The Massachusetts Institute of Technology (MIT) developed an implementation of both GSS-API and its Kerberos mechanism. It was unfortunately regulated by US export restrictions since it was classified as auxiliary military technology, due to its use of the Data Encryption Standard (DES) encryption algorithm. This prompted a non-US entity, the Royal Institute of Technology in Sweden, to develop an open source alternative, which they named Heimdal. The initiative encompassed implementations of pure Kerberos 5, NTLM, GSS-API, SPNEGO, and authentication mechanisms using Kerberos and NTLMSSP. The remainder of this subsection looks into the internals of the Heimdal suite of protocols.

The GSS-API implementation detects the mechanisms available on the system through a well-defined configuration file, statically named and residing in `/etc/gss/mech`. In other implementations, this could be configured via environment variables. An example file entry is located in Code Snippet 2.3. It consists of newline separated mechanism entries. Each entry is divided into four whitespace-separated components; the first column contains the plain-text name of the back-end security mechanism implementing GSS-API, the second holds the OID, and the third column names the shared library of the mechanism. The fourth column is an optional component referencing the kernel module implementing the service, if any.

Code Snippet 2.3 : An example of the contents of a `/etc/gss/mech` file.

```
# Name          OID                     Shared Library          Kernel module
# ------------------------------------------------------------------------
  spnego        1.3.6.1.5.5.2           libgssapi_spnego.so.10  -
  ntlm          1.3.6.1.4.1.311.2.2.10  libgssapi_ntlm.so.10    -
  kerberosv5    1.2.840.113554.1.2.2    libgssapi_krb5.so.10    kssapi_krb5
```

Each entry is parsed and represented in a custom structure. The shared library specified in the third column of each mechanism entry is opened in local scope through `dlopen()`. The symbol `_gss_name_prefix` is resolved with `dlsym()`, which when invoked returns a string constant specific to the mechanism library. The GSS-API library will use this string constant as a prefix to all other well-known methods, resolving their symbols with `dlsym()`. This allows the facilitating GSS-API library to redirect function calls into the designated mechanism.

3

Protocol

SMB, regardless of version, provides a networked protocol between a server and client. It offers three services:

- Access to files and directories of an exposed directory root on the server. It supports locking, encryption, notification alerts etc.
- Issue Remote Procedure Call (RPC) operations to services residing on the server machine.
- Operate network attached printers administrated by the server.

This chapter serves as a guide through the depths of the SMB2 protocol and attempts to introduce some order to the vague and partly inconsistent nature of the technical document defining SMB2 [11]. Any reference to SMB or the 'protocol' refer only to the technical document defining SMB2 [11], unless it is explicitly stated to be valid or part-of the SMB1 standard [30].

3.1 History

The initial draft of the SMB1 protocol originated from IBM with the goal of turning local file access into a networked file system. Microsoft adopted the most commonly used version and heavily modified it, turning it into their own

proprietary protocol with no official documentation of its semantics. Reverse engineering efforts of available implementations from corporations like DEC and Microsoft started early. Most notable is the Samba open source project for Unix. Throughout the nineties, the major development and use of SMB was driven by Microsoft and their family of operating systems. Without any technical reference or standard available, Microsoft dictated the evolution of SMB for multiple years in the mid-nineties. At one milestone in 1996, they launched an initiative to re-brand SMB to Common Internet File System (CIFS) [31]. They submitted a IETF working draft of CIFS, which yielded valuable insight for the open source project, but has since expired [32].

The initial SMB protocol ran over NetBIOS, most notably NBF introduced in Section 2.1. This is evident from multiple structure fields in SMB1 commands referencing NetBIOS components. SMB was later adapted to run on transports such as NetBIOS over IPX/SPX (NBX) and NBT, before being effectively deprecated with the announcement of CIFS with the ability to run on a "raw" TCP connection. One issue arose with this switch of transports; NetBIOS sessions contained the total packet length whilst TCP is a continuous stream deprived of this functionality. Therefore, each packet sent over a raw TCP connection require a four byte header, known as the NetBIOS Session Service Header. Three out of the four bytes present in the NetBIOS Session Service Header encode the total packet length, leaving the last byte to represent the NetBIOS command code which must be zero. The byte encoding of this header is in Big-Endian (network order), whilst the remainder of SMB uses Little-Endian byte ordering.

Even though Microsoft was the main driving force behind SMB, multiple other corporations ran the protocol. For partial or complete interoperability with regards to version and capabilities, a specific dialect was always negotiated as part of establishing a session. In negotiation of SMB1, an array of ASCII encoded strings represents supported dialects of the client. The server responds with an index into this dialect array to pick its preferred protocol version, or terminate the connection if no common dialect is found. Clients can advertise their support of SMB2 by including the string "SMB 2.002", whereupon the server responds with a SMB2_NEGOTIATE command to indicate its capability to continue protocol negotiation over SMB2. An example packet trace of an SMB1 negotiate request can be found in Packet Trace 3.1. The SMB versions are differentiated by the initial four bytes of the header, which contain a magic identifier. For SMB1, the contents of these bytes are "\xFFSMB", whilst SMB2 use "\xFESMB". The client detects the version upgrade by these magic identifier bytes. Because of this capability to negotiate a switch from SMB1 to SMB2 over the same raw TCP connection, SMB2 is required to keep the NetBIOS Session Service Header instead of other alternatives of representing total packet length.

Packet Trace 3.1 : An example SMB1 negotiate request. The structure of every SMB1 request is divided into three fields; header, word array and byte array. As exemplified by this negotiation request, the byte array contains null-separated ASCII strings for each supported dialect by the client. The presence of both "SMB 2.002" and "SMB 2.???" indicates that the client support SMB2, and multiple dialects thereof.

SMB Header

Magic Identifier:	"\xFFSMB"
SMB Command:	Negotiate Protocol (0x72)
NT Status:	STATUS_SUCCESS (0x0)
Flags:	0x18
Flags2:	0xC835
Process ID High:	0
Signature:	0000000000000000
Reserved:	0000
Tree ID:	0
Process ID:	65279
User ID:	0
Multiplex ID:	0

Negotiate Protocol Request

Word Count:	0
Byte Count:	120
BufferFormat:	0x02 (Dialect)
Dialect:	PC NETWORK PROGRAM 1.0
Dialect:	LANMAN1.0
Dialect:	Windows for Workgroups 3.1a
Dialect:	LM1.2X002
Dialect:	LANMAN2.1
Dialect:	NT LM 0.12
Dialect:	SMB 2.002
Dialect:	SMB 2.???

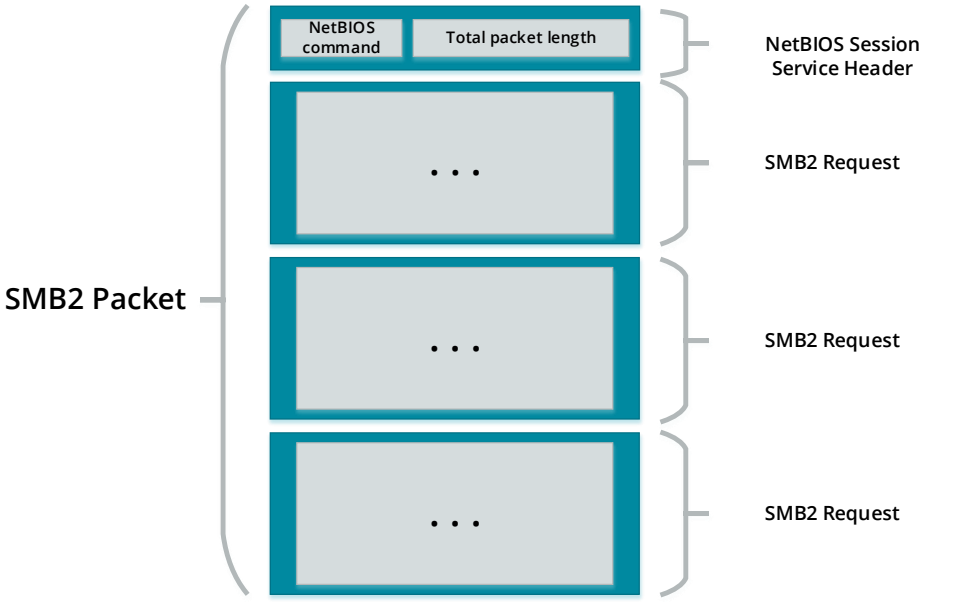


Figure 3.1: Layout of an SMB2 Packet.

There were many caveats with the original SMB protocol, much in regard to its "chattiness" and it being devoid of latency concerns in its design. Menial tasks require a series of synchronous network round trips to complete. It was not designed for Wide Area Networks (WANs) or for high latency networks, which limited the use of compounding multiple commands into a single network packet. It evolved and mutated through dialects to retrofit functionality it was never envisioned to support, such as unicode, and it was self-constrained with regard to number of open files, shares or active users, since it used 16 bit identifiers for such tasks. With the large number of commands and arbitrary (small) constraints on key features, the protocol was inherently difficult to extend, maintain, and secure [33].

3.1.1 Introducing SMB2

SMB2 was first introduced by Microsoft in Windows Vista [34]. Even though this new version of SMB carries the same name as its predecessor, it bears little to no resemblance to the original. The command set was reduced from well over a hundred to just nineteen [35]. The only resemblance is the NetBIOS Session Service Header and the magic identifier of the header structure, to remain compatible over the same transport. SMB2 is designed with 64 bit architectures in mind, and all structures are properly aligned such that no compiler-specific padding should occur between them.

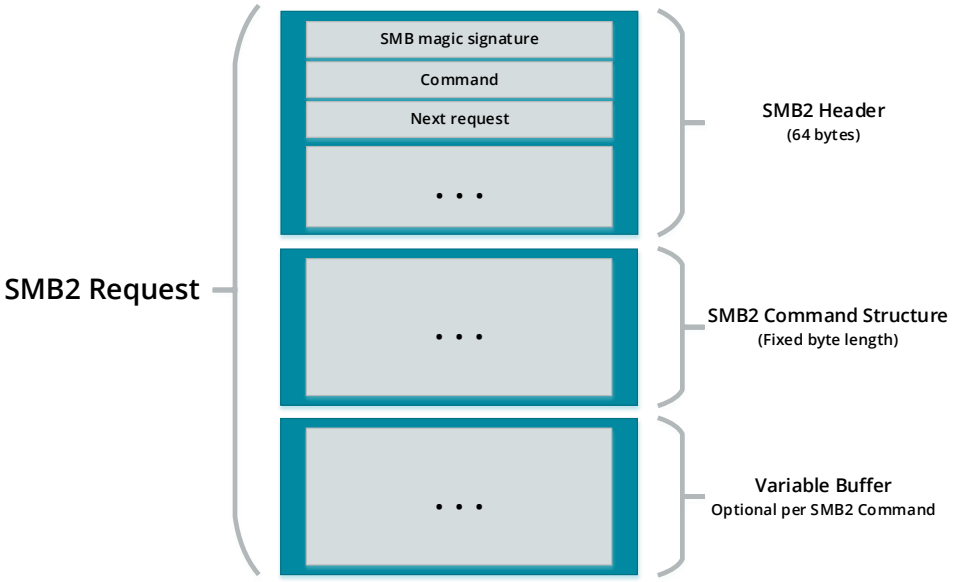


Figure 3.2: Layout of an SMB2 Request.

An SMB2 request can be split into three natural parts; header, command structure, and a variable-sized buffer. This is illustrated in Figure 3.2. Common for all requests is the 64 byte fixed size header. It comes in two varieties; synchronous and asynchronous. Figure 3.3 depicts their layout, whose only difference is the 8 bytes occupying the byte-range from 32 to 40. The headers are differentiated by a bit in the **Flags** field named ASYNC_COMMAND, that when set indicates that the request may be processed asynchronously. The 2-byte **Command** field of the header identifies which operation this request concerns. Each command is of fixed size, varying dependent upon which type it is. Some of these commands contain a variable-sized buffer with additional information, whose length and offset is specified in the command structure.

There are two noninterchangeable ways to compound multiple requests into a single packet. Figure 3.1 shows the composition of a packet, with the NetBIOS session service header followed by one or more requests. The **Next Command** field of the header specifies an 8-byte aligned offset from the beginning of the current request to the start of the next one. The remaining bytes between the end of the current request to the next one are padded with zeros until the 8-byte boundary. If the series of requests are related and require serialization, a bit known as RELATED_OPERATIONS in the **Flags** header field must be set. This flag must only be set for request number two and out in the chain, and may not be omitted. This requires a synchronous request. If one of the commands in the chain results in a fault, the remaining requests generate a similar fault and are not properly processed. An example of such a compound

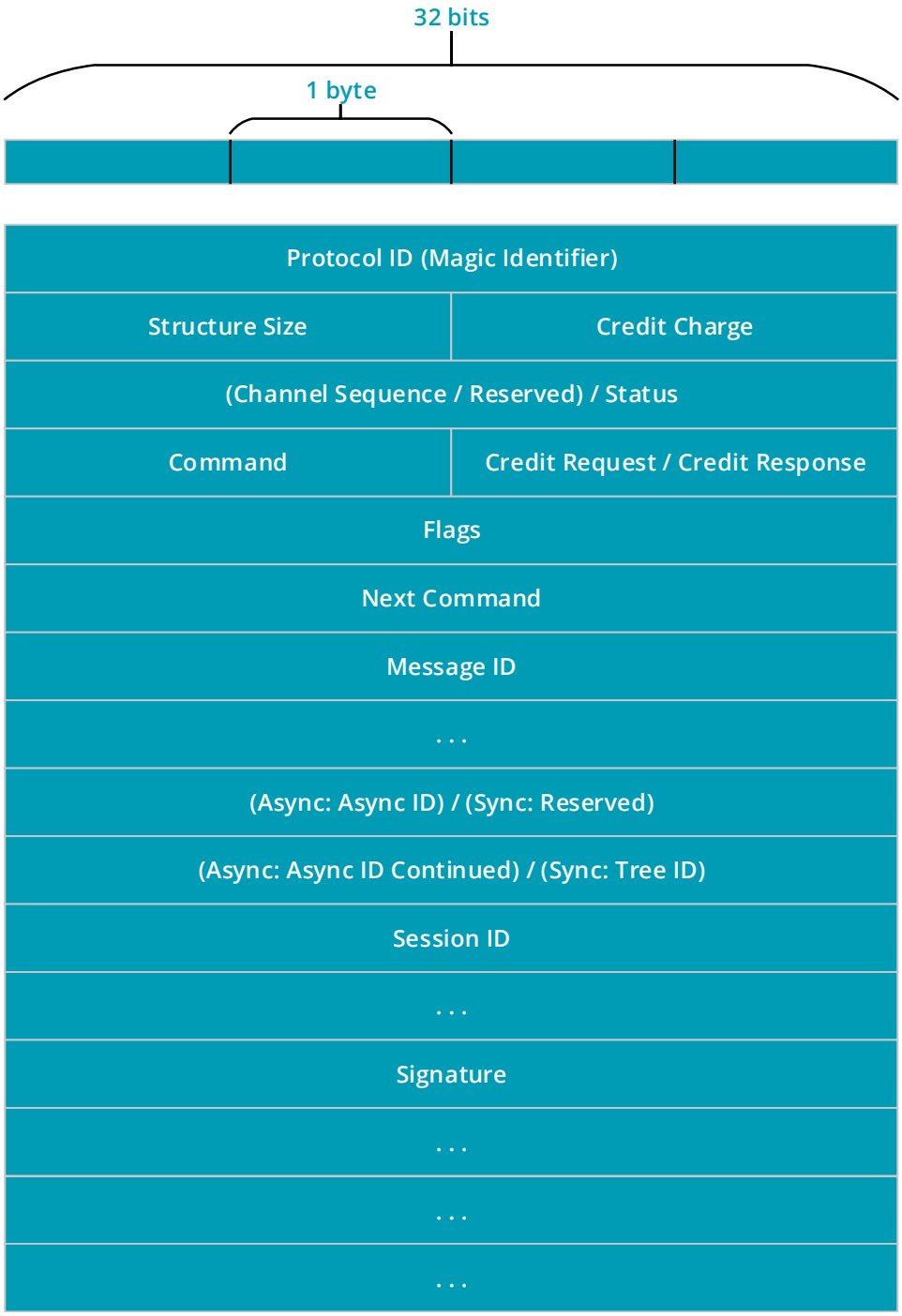


Figure 3.3: Layout of the SMB2 header.

chain of related operations is to open a file, read from it, and close it. If the RELATED_OPERATIONS flag is not set from the second request and onwards, the requests are totally unrelated and may be processed independent and asynchronously. These two methods of specifying compound requests must not be used interchangeably. The server is required to handle compound requests from the client, but may chose to reply to each request individually.

It is worth noting that the text encoding used in SMB2 for string-based fields are exclusively Unicode, specifically in UTF-16 encoding. Support for this encoding is required to run the protocol. None of the fields are null-terminated, unless explicitly stated otherwise.

The transports available to SMB2 are not limited to raw TCP/IP on port 445, which was introduced with CIFS for SMB1, but also include transports such as Remote Direct Memory Access (RDMA) [36]. The implementations of RDMA supported include RDMA over Converged Ethernet (RoCE) [37], iWARP [38][39] and InfiniBand Reliable Connected mode [40]. The interaction of RDMA with SMB2 is detailed in the technical document from Microsoft named "SMB2 Remote Direct Memory Access (RDMA) Transport Protocol" [36].Due to backwards compatibility constraints, the legacy NetBIOS transport is still officially supported by the protocol; however, it is not required for interoperability with clients in the wild, since these usually start off attempting to establish a raw TCP connection.

At the time of writing, SMB2 has four dialects. SMB2 2.002 is the original dialect, released with the introduction of SMB2 in Microsoft Vista. Dialect 2.1 was introduced in Windows 7, yielding minor performance enhancements with an opportunistic locking mechanism. SMB 3.0, inexplicably bumped to a new major version without actually defining a new protocol, facilitates the RDMA transport explained above. It also introduced a channel abstraction, allowing multiple connections per session. SMB 3.02 is the latest released dialect, adding only minor improvements.

3.1.2 Commands

SMB2 supports 19 commands, down from over a hundred in SMB1. They each come with a unique protocol frame that occupy the command structure part of Figure 3.2, ranging in size from just 4 bytes in SMB2_ECHO to 89 bytes in the response structure of SMB2_CREATE. All 19 commands are listed in Table 3.1. The first column lists the command names in plain English, whilst the second column prints the textual reference used to uniquely identify and refer to the SMB2 commands throughout this thesis. The last column shows the 16-bit value each command is identified by, which is the **Command** field

of the SMB2 header illustrated in Figure 3.3.

All commands have an unqiue structure for both the client request and a server response, which need not be equal. As per old Microsoft style of protocol creation, each structure begins with a 16-bit **StructureSize** field to hold the total size of the structure. Whenever this field diverges from expected value, the processing of the request halts and the entire operation is returned with an invalid parameter status code. If the structure size is an odd value, the command is succeeded by a variable length buffer whose content and size must be individually interpreted by members of said structure. If the length indicator specifies zero bytes in this buffer, a zero valued byte must still be present.

Communication is client-driven; it initiates a request and expects a response from the server. The only exception is SMB2_OPLOCK_BREAK, where the server may break a previously established lock held by the client. This request is issued by the server with a notification, whereupon the client acknowledges the operation and finally the server responds to the client acknowledgment.

The following subsections exemplify some of the most vital operations and their relation to each other.

Command: Negotiate

SMB2_NEGOTIATE is the first command exchange on a new transport. Any other **Command** identifier in the SMB2 header for when **Message_id** is zero, which is the first request, is considered invalid and the connection is terminated. The command request structure is depicted in Figure 3.4. The primary use of SMB2_NEGOTIATE is to exchange initial security mode settings and arrive on a common supported dialect between client and server. An example extract of this request is found in Packet Trace 3.2. Note that the length of the variable buffer is not specified in the command structure, but rather calculated from the **Dialect Count** field. Each **Dialect** is 16-bits, such that the total variable buffer length is **Dialect Count** multiplied by two.

After processing a SMB2_NEGOTIATE request, the server issues a similar response structure, depicted in Figure 3.5. The server picks a dialect from those listed in the **Dialects** variable field of the request, and echoes this in the **Dialect Revision** of the response. Various security options and capabilities are relayed back alongside simple meta-data about the server itself.

Perhaps the most interesting part of this command exchange is the contents of the variable sized **Security Buffer**, which is described by the fields **Security**

Command	Text reference	Value
Negotiate	SMB2_NEGOTIATE	0x0000
Session Setup	SMB2_SESSION_SETUP	0x0001
Logoff	SMB2_LOGOFF	0x0002
Tree Connect	SMB2_TREE_CONNECT	0x0003
Tree Disconnect	SMB2_TREE_DISCONNECT	0x0004
Create	SMB2_CREATE	0x0005
Close	SMB2_CLOSE	0x0006
Flush	SMB2_FLUSH	0x0007
Read	SMB2_READ	0x0008
Write	SMB2_WRITE	0x0009
Lock	SMB2_LOCK	0x000A
IOCtl	SMB2_IOCTL	0x000B
Cancel	SMB2_CANCEL	0x000C
Echo	SMB2_ECHO	0x000D
Query Directory	SMB2_QUERY_DIRECTORY	0x000E
Change Notify	SMB2_CHANGE_NOTIFY	0x000F
Query Info	SMB2_QUERY_INFO	0x0010
Set Info	SMB2_SET_INFO	0x0011
OPLock Break	SMB2_OPLOCK_BREAK	0x0012

Table 3.1: Table of all SMB2 commands and their corresponding identifier value.

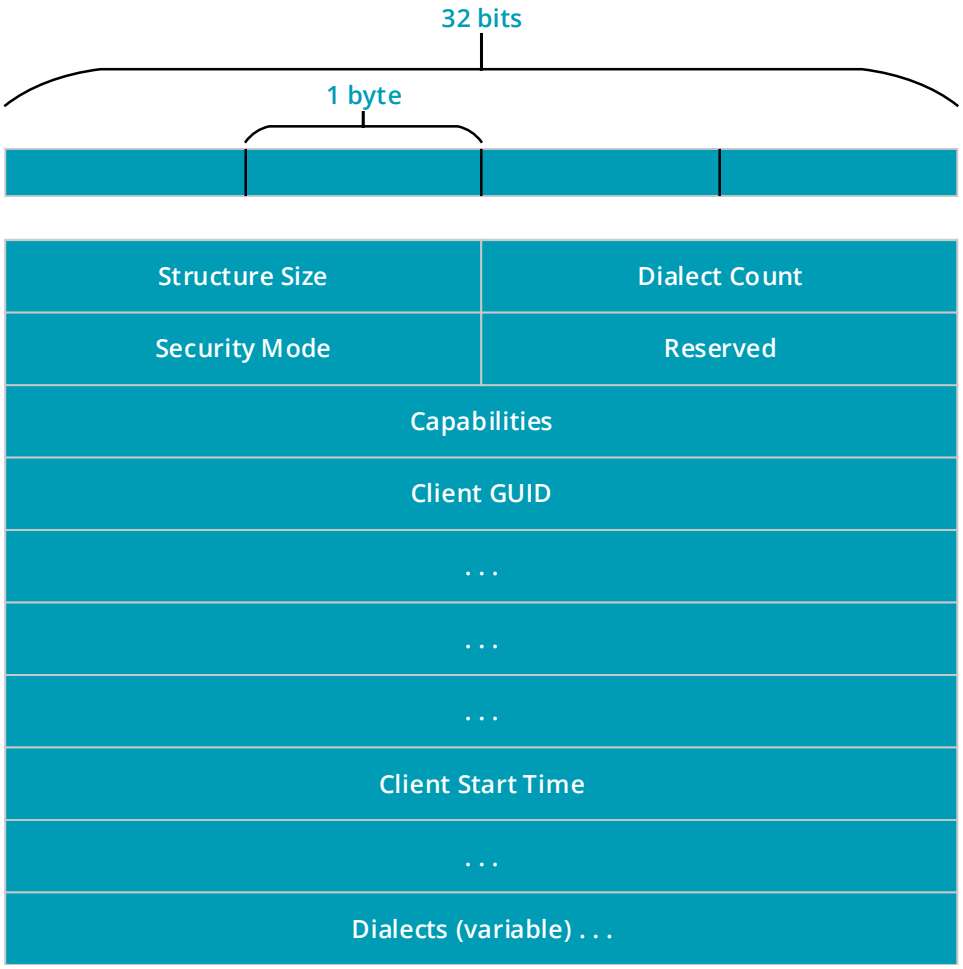


Figure 3.4: Layout illustration of the SMB2_NEGOTIATE request command structure. The length of the variable field is a two byte multiplier of the **Dialect Count** field, since each dialect identifier is 16-bits long.

Packet Trace 3.2 : An example packet trace for a SMB2_NEGOTIATE request. This trace is extracted from a Windows 8.1 SMB2 client.

```
SMB2 Header
    Magic Identifier:       "\xFESMB"
    Structure Size:         0x0040
    Credit Charge:          0
    NT Status:              STATUS_SUCCESS (0x000000)
    Command:                Negotiate (0x00)
    Credits Granted:        1
    Flags:                  None, (0x00000000)
    Chain Offset:           0x00000000
    Message ID:             0
    Reserved:               0
    Tree ID:                0x00000000
    Session ID:             0x0000000000000000
    Signature:
        0x00000000000000000000000000000000
Negotiate Protocol Request
    Structure Size:         0x0024
    Dialect Count:          4
    Security Mode:          0x01
      Enabled               1: True
      Required              0: False
    Capabilities:           0x0000007F
      DFS                   1, Host supports DFS
      Leasing               1, Host supports LEASING
      Large MTU             1, Host supports LARGE MTU
      Multi Channel         1, Host supports MULTI
          CHANNEL
      Persistent Handles    1, Host supports PERSISTENT
          HANDLES
      Directory Leasing     1, Host supports DIRECTORY
          LEASING
      Encryption            1, Host supports ENCRYPTION
    Client GUID:
        7EDA47E5-D5C8-11E4-8285-3417EB9AE8EA
    Boot Time:              Unspecified (0x00000000)
    Dialects:
      Dialect               0x0202
      Dialect               0x0210
      Dialect               0x0300
      Dialect               0x0302
```

Buffer Offset and **Security Buffer Length**. The technical documents use the RFC-2119 definitions of MUST, SHOULD, MAY and their variants [41], which infers that the **Security Buffer** SHOULD contain a preemptive GSS token to use server-initiated SPNEGO authentication (described in Section 2.3). However, it also states that if the **Security Buffer Length** is zero, the variable buffer is empty and client-initiated authentication with a protocol of the clients choice will be used instead. By the definition of SHOULD, there may exist valid reasons to ignore a particular item, and this certainly seems to qualify to such an exception since the protocol describes an alternative approach. A server may not neglect to return an opportunistic token and let the authentication initiative be client-driven if it hopes to remain compatible with the real world. Existing clients, such as Samba and Windows, outright terminate the connection if no opportunistic token is received. This is a clear testimony to the discrepancies between the intended mechanism of the protocol and how the real world handles the situation. A complete description of the protocol security mechanism is available in Section 3.3.

3.2 Central structures

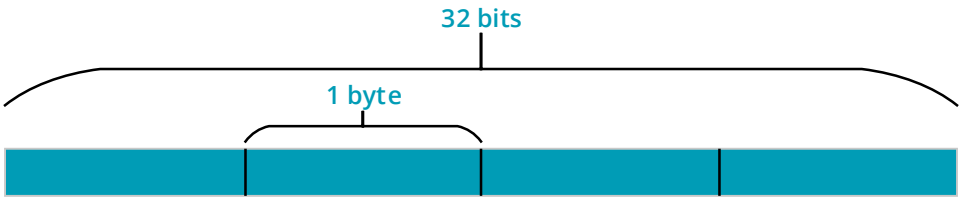
The technical document, section 3.2 and 3.3, contains two exhaustive subsections detailing the internal composition of both server and client components, alongside explanations for processing a myriad of situations that may occur. Both the central structures, their member components and naming, are merely advisory for the technical document, but serve equally well as a baseline for anyone wishing to implement one or both sides of the protocol.

This section explains the central server structures, with their members and functionality in lightweight terms. The structure listings with associated members are incomplete, but serve as a guide for component responsibilities as explained in the accompanying member comments.

The structures introduced are respectively; **Server**, **Connection**, **Packet**, **Request**, **Session**, **Share**, **Tree_connect**, and **Open**. They all relate to each other and the structures either encapsulate or build upon each other, but some cross-references may occur prior to the expanding explanation.

3.2.1 Structure: Server

The **Server** structure is the root state hub for the server endpoint. It is instantiated upon creation and it is a unique construct with only one instance. A representation of this server construct is found in Structure 3.1.



Structure Size	Security Mode
Dialect Revision	Reserved
Server GUID	
. . .	
. . .	
. . .	
Capabilities	
Max Transaction Size	
Max Read Size	
Max Write Size	
System time	
. . .	
Server Start Time	
. . .	
Security Buffer Offset	Security Buffer Length
Reserved	
Security Buffer (variable) . . .	

Figure 3.5: Layout of the SMB2_NEGOTIATE response command structure.

The construct can maintain several open listen sockets corresponding to different Network Interface Cards (NICs) available to the server, whilst representing them as an array of resource identifiers in **SERVER→LISTEN_SOCKETS**. The underlying mechanics to handle Input / Output (I/O) between accepted clients and the server is handled through an *asynchronous I/O* engine, represented by the **SERVER→ENGINE** member of Structure 3.1.

The most prominent tasks of the **SERVER** is to keep global state over resources, provide registration services, guarantee unique file access, and the like. For instance, each client, session, and open are queriable by identifiers through a globally available hashmap. The reminder of the members serve the idle task of holding state variables, capabilities, and journal statistics about the performance on the server.

Some of the data types defined are Abstract Data Types (ADTs) specific to a given implementation, but the associated comment should clarify the intent of its member. For instance, the WAUTH_SERVER_CONTEXT_T ADT is present to illustrate the presence of a server-side authentication component.

3.2.2 Structure: Connection

The **CONNECTION** structure is responsible of representing a single connected client over a given transport. A representation, adapted to raw TCP only, is found in Structure 3.2.

SERVER→ENGINE handles all communication for both read and write operations. The resource identifiers and VXL_SAIOPIPE_T abstraction interface with the engine to provide these services. All structure members prefixed with CLIENT_, relating to security mode, capabilities, and identifier, are retrieved through the SMB2_NEGOTIATE commands exchanged when the connection was established.

CONNECTION→CREDITS regulate the number of outstanding requests the client is allowed to have simultaneously on the transport. The credit quota is mirrored in the available **Message_id** accepted by the server, which is mapped to a sequence of unused identifiers in an implementation defined ADT, **CONNECTION→MESSAGE_SEQUENCE_NUMBER**.

If the client reaches zero credits or provides a **Message_id** that is not present in **CONNECTION→MESSAGE_SEQUENCE_NUMBER**, the request will not be processed, and the connection is brought to a halt.

Finally, **CONNECTION→SESSION_TABLE** holds a map of all established ses-

Structure 3.1 : Example of structure members useful for an SMB2 Server component.

```
struct Server
{
    //! Array of listen sockets available to the Server.
    vx_rid_t                    listen_sockets[MAXIMUM_AVAILABLE_NETWORK_OBJECTS];
    //! AIOEngine associated with the Server. All I/O operations are performed through this.
    vxl_aioengine_t             *engine;
    //! Authentication ADT - store everything relevant for the authentication
    //! mechanisms of the Server through this wauth handler
    wauth_server_context_t      *auth;


    //! Whether or not the Server will be accepting incoming connections or requests.
    vx_bool_t                   enabled;
    //! Structure registering statistics about the Server ( see MS-SRVS section 2.2.4.39 )
    statistics_t                statistics;


    //! Global unique identifier for this Server generated upon Server creation.
    vx_uint8_t                  guid[SMB2_CLIENT_GUID_LENGTH];
    //! Start time of the SMB2 Server in FILETIME format.
    filetime_t                  start_time;


    //! If set, indicates that the Server supports Distributed File System.
    vx_bool_t                   is_dfs_capable;
    //! Indicates whether the Server requires messages to be encrypted after session establishment
    vx_bool_t                   encrypt_data;
    //! Indicates whether the Server will reject any unencrypted messages.
    //! Only applicable if Server->encrypt_data or Share->encrypt_data is set.
    vx_bool_t                   reject_unencrypted_access;
    //! Indicates whether the Server REQUIRE messages to be signed if the
    //! Connection is neither anonymous nor guest.
    vx_bool_t                   require_message_signing;


    //! List of all available shares registered for this Server.
    share_t                     *share_list;
    //! List of all active Sessions established to this Server, indexed by Session->session_id
    vxl_hashmap_t               *global_session_table;
    //! Table of all Opens by remote Clients on the server indexed by Open->durable_file_id
    vxl_hashmap_t               *global_open_table;
    //! Table of all Connections to the server indexed by Connection->connection_id
    vxl_hashmap_t               *connection_table;


    //! Maximum number of chunks the Server will accept in a server-side copy operation.
    vx_uint64_t                 copy_max_number_of_chunks;
    //! Maximum number of bytes the Server will accept in a single chunk
    //! for a server-side copy operation.
    vx_uint64_t                 copy_max_chunk_size;
    //! Maximum number of bytes the Server will accept for a server-side copy operation.
    vx_uint64_t                 copy_max_data_size;


    //! Counter for ever increasing Connection->connection_id, to keep them unique.
    vx_uint64_t                 allocated_connection_ids;
    //! Counter for ever increasing Session->session_id, to keep them unique.
    vx_uint64_t                 allocated_session_ids;
    //! Counter to allocated unqiue Open->durable file_id for the Server.
    vx_uint64_t                 allocated_durable_file_ids;
} server_t;
```

sions on the transport, indexed by **Session_id** supplied in the header of each request (except for SMB2_NEGOTIATE and SMB2_SESSION_SETUP).

3.2.3 Structure: Packet

The **PACKET** structure encapsulates the processing of an entire packet, possibly compounded into multiple requests, and the responses of the processed requests. The server may chose to send a response packet for each of the compounded requests, or compound the responses as well. VXL_CBUFFER_T is the OS buffer abstraction for both incoming and outgoing packets. It hides the underlying connection transport from the request handling. Retrieving and inputting data through this interface aid the development process with multiple safeguards. For instance, it detects buffer-overflows and reads past boundaries. Writing to **PACKET→OUTGOING_BUFFER** centralizes the computation of the total packet length part of the NetBIOS Session Service Header.

The series of **PACKET→COMPOUND_** members, which hold **File_id**, **Session_id**, and **Tree_id**, are essential to this structure and may not reside elsewhere. They act as an up-to-date storage of identifiers that may be valid across multiple requests, and not provided by each request individually. SMB2 supports multiple related requests compounded into a single packet, each dependent upon the successful processing of the prior request in the chain. For instance, given a packet of three compound related requests, SMB2_CREATE, SMB2_READ, SMB2_CLOSE in that order, the last two requests depend upon the **File_id** assigned in the first processed SMB2_CREATE operation. Both command structures contain a field to supply this **File_id**, but cannot possibly know the identifier assigned to the **OPEN** by the SMB2_CREATE operation. Thus, each operation that results in the creation or opening of an identifier by a previous request must refresh the content of their respective **PACKET→COMPOUND_** identifier. This allows the last two requests to access the **File_id** to perform their respective operations, all compounded into a single packet.

3.2.4 Structure: Request

The **REQUEST** structure contains all information necessary to receive, process, and respond to a client request. A reference representation is found in Structure 3.4. This contains the SMB2 header, command, and variable buffer structures for both the incoming client request and the outgoing server response.

REQUEST→RESPONSE_STATUS_CODE is populated with an appropriate

Structure 3.2 : Example content of a structure used to keep state of a client Connection.

```
struct Connection
{
    //! Resource identifiers for the accepted socket Connection.
    vx_rid_t                        read_rid;
    vx_rid_t                        write_rid;

    //! Dedicated input/output pipes to the socket associated with read_rid/write_rid.
    vxl_saiopipe_t                  *read_pipe;
    vxl_saiopipe_t                  *write_pipe;

    //! Packet currently being processed by the Connection.
    packet_t                        *packet;
    //! Backwards reference to the running Server structure.
    server_t                        *server;


    //! Uniquely identifies a Connection on the Server.
    vx_int64_t                      connection_id;

    //! Null-terminated unicode IP-address or NetBIOS host name of the Client machine.
    vx_utf8_t                       client_name[SMB2_CONNECTION_CLIENT_NAME_LENGTH];
    //! Timestamp for when the Connection was established in FILETIME format.
    filetime_t                      client_established_time;
    //! Capabilities of the Client on this Connection. See section 2.2.3 for the syntax.
    smb2_capabilities_t             client_capabilities;
    //! An identifier for the Client machine.
    vx_utf8_t                       client_guid[SMB2_CLIENT_GUID_LENGTH];


    //! Available credits for this client.
    vx_uint64_t                     credits;
    //! ADT member to handle valid sequence numbers to receive from this Connection.
    message_sequence_number_t       *message_sequence_number;


    //! Map of all requests being processed synchronously by the server over this Connection.
    //! Indexed by Request->message_id
    vxl_hashmap_t                   *request_list;
    //! Map of all asynchronously processing requests on this Connection.
    //! Indexed by Request->async_id
    vxl_hashmap_t                   *async_command_list;
    //! Map of authenticated sessions for this connection.
    //! Indexed by Session->session_id.
    vxl_hashmap_t                   *session_table;


    //! Indicates that all sessions on this connection must have signing enabled.
    //! Anonymous and Guests sessions are exempt from this requirement.
    vx_bool_t                       should_sign;
    //! Indicating whether the connection supports multi-credit operations.
    vx_bool_t                       supports_multi_credit;


    //! Current state of the dialect negotiation between Server and Client.
    vx_uint32_t                     negotiate_dialect;
    //! The SMB2 negotiated dialect of which to communicate with the Client.
    vx_uint32_t                     dialect;
    //! Indicates whether or not the authentication of a
    //! non-anonymous principal has not yet been established.
    vx_bool_t                       constrained_connection;


    //! Maximum buffer size in bytes the Server allows on the transport for the following operations
    //! QUERY_INFO, QUERY_DIRECTORY, SET_INFO and CHANGE_NOTIFY.
    vx_uint64_t                     max_transaction_size;
    //! Maximum buffer size in bytes the Server allows to be written on
    //! the connection during an SMB2_WRITE request.
    vx_uint64_t                     max_write_size;
    //! Maximum buffer size in bytes the Server allows to be read on
    //! the connection during an SMB2_READ request.
    vx_uint64_t                     max_read_size;


    //! The Server capabilities established in the SMB2_NEGOTIATE_RESPONSE.
    smb2_capabilities_t             server_capabilities;
    //! The Client security mode sent in SMB2_NEGOTIATE_REQUEST.
    smb2_security_mode_t            client_security_mode;
    //! The Server security mode of sent to the Client in SMB2_NEGOTIATE_RESPONSE.
    smb2_security_mode_t            server_security_mode;
} connection_t;
```

Structure 3.3 : A sample structure containing members to keep track of a single packet arriving on a Connection transport.

```
struct Packet
{
    //! Buffer containing the incoming packet data from a Connection.
    vxl_cbuffer_t       *incoming_buffer;
    //! Buffer to write a response packet.
    vxl_cbuffer_t       *outgoing_buffer;

    //! Pre-allocated request_t structures to handle 5 compound requests
    //! without dynamically allocating memory upon each compound request packet received.
    request_t           requests[5];
    //! Backup storage capacity if the requests array is filled
    //! and we have additional compound requests in packet.
    request_t           *additional_requests;

    //! Number of requests sent in this packet.
    vx_uint16_t         num_compound_requests;
    //! Number of requests finished all processing and reply is buffered.
    vx_uint16_t         processed_requests;


    //! Total packet length - Extracted from the first 4 bytes in the packet
    //! from the NetBIOS session service header.
    vx_uint32_t         incoming_packet_length;
    //! Total length of all outgoing requests - does not include the NetBIOS Session Serivce length.
    vx_uint32_t         outgoing_packet_length;
    //! Total number of bytes read from the incoming packet buffer during request processing
    vx_uint32_t         bytes_read_from_packet;


    //! If the request is part of a compound operation, the file_id for
    //! the entire series of requests will be stored up-to-date here.
    smb2_file_id_t      compound_file_id;
    //! If the request is part of a compound operation, the session_id for
    //! the entire series of requests will be stored up-to-date here.
    vx_int64_t          compound_session_id;
    //! If the request is part of a compound operation, the tree_id for
    //! the entire series of requests will be stored up-to-date here.
    vx_uint32_t         compound_tree_id;


    //! Backwards reference to the connection handling this packet
    connection_t        *connection;
} packet_t;
```

status code during processing, which is propagated to the final response dispatch. The NTSTATUS error codes are defined in MS-ERREF [42].

Structure members such as **Command**, **Message_id**, and **Compound_request**, are present as an optimization and could be retrieved from their respective position in **REQUEST→REQUEST_HEADER**.

Structure 3.4 : A sample structure containing all members necessary to process a single request/command.

```
struct Request
{
    //! Union type to keep variables required across multiple processing layers
    //! within the single command parsing, verification and execution.
    smb2_command_processing_variables_t                 command_variables;

    //! Copy of the incoming request SMB2 header retrieved from Packet->incoming_buffer
    smb2_header_t                                       request_header;
    //! Union type holding all the associated request command structures.
    smb2_request_command_t                              request_command;
    //! The variable length buffer content of each command MAY be parsed into this union location.
    //! Command dependent.
    smb2_command_request_variable_buffer_content_t      request_variable_buffer_content;
    //! Number of bytes read from packet associated with this request.
    vx_uint32_t                                         request_bytes_read_from_packet;

    //! Populated response header, copied into the reply buffer upon write.
    smb2_header_t                                       response_header;
    //! Error response structure - Returned upon a Request->response_status_code different than
        STATUS_SUCCESS.
    smb2_error_response_t                               response_error;
    //! Union type holding all the associated response command structures.
    smb2_response_command_t                             response_command;
    //! The variable length buffer holding data to be supplied in the variable_buffer command
        response.
    smb2_command_response_variable_buffer_content_t     response_variable_buffer_content;
    //! Length in bytes of the response written into the outgoing buffer.
    vx_uint32_t                                         response_length;
    //! Status code returned in Request->response_header
    NTSTATUS                                            response_status_code;

    //! The command identifier of this Request from the Request->request_header
    vx_uint16_t                                         command;
    //! Message id from the Request->request_header.
    vx_uint64_t                                         message_id;
    //! Generated async id if the Request is to be processed asynchronously
    vx_uint64_t                                         async_id;

    //! Boolean set if this request is part of handling a compound Request,
    //! and is not the first Request in the compound chain.
    vx_bool_t                                           compound_request;


    //! Reference to a Session this request MAY be associated with.
    //! NULL if no such association exists.
    session_t                                           *session;
    //! Reference to the TreeConnect this Request MAY be associated with.
    //! NULL if no such association exists.
    tree_connect_t                                      *tree_connect;
    //! Reference to an Open this request MAY be associated with.
    //! NULL if no such association exists.
    open_t                                              *open;

    //! Reference to the Connection this request is transfered on.
    connection_t                                        *connection;
    //! Backwards reference to Packet handling structure from which this Request originated.
    packet_t                                            *packet;

    //! Ability to chain outstanding Requests;
    request_t                                           *next, *prev;
} request_t;
```

3.2.5 Structure: Session

The **Session** encapsulates an established session between a client **Connection** and the server. Most SMB2 commands are only valid as part of an authenticated session, which is established through a series of SMB2_SESSION_SETUP command exchanges. Structure 3.5 represents a server-side definition.

When the client establishes a new session, the server will allocate a new **Session** structure and generate an unqiue identifier assigned to the session. This is stored in **Session→session_id**. The authentication context through GSS-API is represented by an ADT, **Session→auth**. After is successful establishment, the context is queried for session keys, user names, and authentication modes which are safely stored in the **Session**.

The **Session** keeps track of all connected shares and open operations performed on the session through hashmaps, which are indexed by their respective **Open→file_id** and **Tree_connect→tree_id**. To keep these identifiers unique, the **Session** is required to keep track of previously allocated identifiers.

3.2.6 Structure: Share

Share represents either of the three supported share types in SMB2, pipe, printer, or disk tree. SMB1 was originally developed to expose a local file system over the network. The **Share**, represented in Structure 3.6, must be registered with an unique name to the server prior to clients attempting to establish a SMB2_TREE_CONNECT to it.

If the **Share** represents a disk tree share type, **Share→local_path** is populated with the local path of the exported directory root. The remainder of its members simply reflect configurable options and state of the **Share** creation.

3.2.7 Structure: TreeConnect

Tree_connect is the structure identifying a successful binding between a client **Session** and a registered server **Share** resource, which is established through a SMB2_TREE_CONNECT command. Its representation is found in Structure 3.7.

Tree_connect→tree_id is a unique identifier allocated by the session and indexes into the **Session→tree_connect_table**, such that the ap-

Structure 3.5 : Sample structure to keep track of a single session established on a transport Connection.

```
struct Session
{
    //! Index into the Server->global_session_table. Unique identifier for this Session.
    vx_int64_t                  session_id;
    //! Current activity state of this Session.
    SMB2_SESSION_STATE_T        state;

    //! Authentication context for this Session.
    wauth_connection_context_t  *auth;

    //! First 16-bytes of the cryptographic key for this authenticated context.
    //! If the key is less than 16 bytes, its right padded with zero
    vx_uint8_t                  session_key[SMB2_SESSION_KEY_LENGTH];
    //! Name of the user who established the Session.
    vx_utf8_t                   user_name[SMB2_SESSION_USER_NAME_LENGTH];

    //! Indicating whether or not the Session is for an anonymous user.
    vx_bool_t                   is_anonymous;
    //! Indicating whether or not the Session is for an guest user.
    vx_bool_t                   is_guest;
    //! Indicating whether or not that all messages for this Session MUST be signed.
    //! This is the defacto authority whether a response will be signed.
    //! Will be calculated in accordance to the values of
    //! Session->is_anonymous, Session->is_guest, share and server state.
    vx_bool_t                   signing_required;

    //! The time the Session was established.
    vx_uint64_t                 creation_time;
    //! A value that specifies the time after which the Client must re-authenticate with the Server.
    vx_uint64_t                 expiration_time;
    //! The time the Session processed its most recent Request.
    vx_uint64_t                 idle_time;

    //! A table of Open that have been opened by this authenticated Session.
    //! Indexed by Open->file_id
    vxl_hashmap_t               *open_table;
    //! A table of Tree_connects that have been established by this authenticated Session.
    //! indexed by Tree_connect->tree_id
    vxl_hashmap_t               *tree_connect_table;

    //! Backwards reference to the Connection on which this Session was established.
    connection_t                *connection;

    //! Counter of allocated Tree_connect->tree_id for this Session - Starts at 1
    vx_uint32_t                 allocated_tree_connect_ids;
    //! Counter for allocated Open->file_id on this Session - Starts at 1
    vx_uint64_t                 allocated_file_ids;
} session_t;
```

Structure 3.6 : Structure with members needed to represent an SMB2 share available on the Server.

```
struct Share
{
    //! Type of share, as per SMB2 specification.
    share_type_t            type;

    //! Name of the shared resource on this server
    vx_utf8_t               name[SMB2_SHARE_NAME_LENGTH];
    //! NetBIOS or textual IPv4 address
    vx_utf8_t               server_name[SMB2_SHARE_SERVER_NAME_LENGTH];
    //! Path that describes the local resource being shared
    vx_utf8_t               local_path[SMB2_SHARE_LOCAL_PATH_LENGTH];


    //! Defined caching policy for this share
    SMB2_CACHING_POLICY_T   caching_policy;
    //! Indicates whether or not this Share is configured for DFS.
    vx_bool_t               is_dfs;

    //! Indicates whether or not the result of directory enumeration
    //! on this Share MUST be trimmed to include only
    //! files and directories the calling user has right to access.
    vx_bool_t               do_access_based_directory_enumeration;
    //! Indicates whether the Clients are allowed to cache directory enumeration results.
    vx_bool_t               allow_client_namespace_caching;
    //! Indicates whether all opens on this share MUST include FILE_SHARE_DELETE in the sharing
        access.
    vx_bool_t               force_shared_delete;
    //! Indicates whether users who request read-only access
    //! to a file are not allowed to deny other reads.
    vx_bool_t               restrict_exclusive_opens;
    //! Indicating that the Server does not issue exclusive caching rights on this share.
    vx_bool_t               force_level2_oplock;
    //! Indicating whether the Share supports hash generation of branch cache retrieval of data.
    vx_bool_t               hash_enabled;


    //! A value indicating the maximum number of concurrent connections to the Share.
    vx_uint16_t             max_concurrent_connections;
    //! A value indicating number of active concurrent connections
    vx_uint16_t             current_concurrent_connections;


    //! Linked list organized - Inserted in the Server->share_list
    share_t                 *next, *prev;
} share_t;
```

propriate **TREE_CONNECT** can be retrieved from the supplied **Tree_id** in the SMB2 header.

Structure 3.7 : Members of a TreeConnect structure

```
struct TreeConnect
{
    //! Numeric value to uniquely identify a TreeConnect within the scope of the Session over which
        it was
    established.
    vx_uint32_t     tree_id;

    //! A numeric value indicating the number of Opens  that are currently active for this
        TreeConnect.
    vx_uint32_t     open_count;

    //! The time this TreeConnect was established.
    vx_uint64_t     creation_time;

    //! Maximal access for the Session to this TreeConnect.
    file_access_t   maximal_access;

    //! Backwards reference to the Session that established this TreeConnect.
    session_t       *session;
    //! Backwards reference to the share that this TreeConnect was established to.
    share_t         *share;
} tree_connect_t;
```

3.2.8 Structure: Open

OPEN represents an successful SMB2_CREATE command to open a resource on an established **TREE_CONNECT**. Its definition is found in Structure 3.8.

OPEN is a vast structure that encapsulates a multitude of primitives to implement the features of SMB2. It contains identifiers for the local resource, the **SESSION→SESSION_ID**, and local file information relating to access rights and file type. The **OPEN** may be opened durable, so it may be re-acquired by the same client after a short network outage. Various levels of locking is supported.

Multiple of the fields hold state information, for instance to query the contents of a directory across multiple requests or the extended attribute list which both could be of arbitrary size.

3.3 Authentication

All services available in SMB2 require the client to establish an authenticated session. There are two commands that facilitate this: SMB2_NEGOTIATE and SMB2_SESSION_SETUP. The latter is the primary method.

Structure 3.8 : Members of the Open structure

```
struct Open
{
    //! Resource identifier for the resource opened
    vx_rid_t            open_rid;

    //! Identifier of the Client operating on this Open.
    vx_utf8_t           client_guid[SMB2_CLIENT_GUID_LENGTH];

    //! Numeric value uniquely identifies the Open handle within the scope of a Session.
    //! This is the volatile handle of a smb2_file_id_t
    vx_uint64_t         file_id;
    //! Numeric value that uniquely identifies the Open handle within the scope of all
    //! Opens granted by the Server.
    //! This is the persistent handle of a smb2_file_id_t
    vx_uint64_t         durable_file_id;

    //! Access granted on this Open.
    file_access_t       granted_access;

    //! Current oplock level for this Open. Must be one of SMB2_OPLOCK_LEVEL_* enumerators.
    SMB2_OPLOCK_LEVEL_T oplock_level;
    //! Current oplock state for this Open. Must be one of the SMB2_OPLOCK_STATE_* enumerators.
    SMB2_OPLOCK_STATE_T oplock_state;

    //! Indicates whether or not this Open was requested durable.
    vx_bool_t           is_durable;
    //! Indicates if this Open is over a directory.
    vx_bool_t           is_directory;
    //! Time value indicating when this durable handle will be closed if the Client does not reclaim
        it.
    vx_uint64_t         durable_open_timeout;
    //! A security descriptor that holds the original opener of the Open.
    //! Allows server to determine if a caller is trying to reestablish a durable open and is
        allowed to do so.
    vx_uint64_t         durable_owner;

    //! Indicates the current location in a directory enumeration, allows for continuing of an
    //! enumeration across multiple requests.
    vx_uint64_t         directory_enumeration_location;
    //! Indicates the search pattern that is usedin directory enumeration, allows for continuing
    //! of an enumeration across multiple requests.
    vx_utf8_t           directory_enumeration_search_pattern[VX_PATH_NVCHAR_MAXNUM];

    //! Indicates the index in the extended attribute information list,
    //! allows enumeration across multiple requests.
    vx_uint64_t         current_ea_index;
    //! Indicates the index in the quota information list, allows enumeration across multiple
        requests.
    vx_uint64_t         current_quota_index;

    //! Numeric value of the number of locks currently held by this Open.
    vx_uint32_t         lock_count;

    //! Variable-length unicode string contains the local path name on the Server that the Open is
        performed on.
    vx_utf8_t           path_name[SMB2_OPEN_PATH_NAME_LENGTH];
    //! Key that identifies a source file in a server-side data copy operation.
    vx_utf8_t           resume_key[SMB2_OPEN_RESUME_KEY_LENGTH];

    //! Indicates whether this Open has requested a resilient operation.
    vx_bool_t           is_resilient;

    //! Reference to the Session on which this Open was performed.
    session_t           *session;
    //! Reference to the TreeConnect under which this Open was performed.
    tree_connect_t      *tree_connect;
    //! Reference to the Connection under which this Open was performed.
    connection_t        *connection;
} open_t;
```

SMB2 relies on the GSS-API, with SPNEGO as its chosen mechanism, to establish a secure context between the client and the server. The sub-protocol performing the authentication is determined by the negotiating SPNEGO mechanism. Chief among them are Kerberos 5 and NTLMSSP. Which authenticating protocol is selected remains invisible to both client and server; it is handled as part of the SPNEGO mechanic and GSS-API protocol.

SMB2_NEGOTIATE response command structure and SMB2_SESSION_SETUP uses its variable length field to transport the binary blob between the respective GSS-API endpoints. As all operations within SMB, authentication is also client-driven. However, the variable length field of SMB2_NEGOTIATE response suggest the server SHOULD supply an opportunistic SPNEGO token. Providing this does not incur any GSS context the server is required to store, since it only serves advisory for the client side who still initiates the context establishment through SMB2_SESSION_SETUP. Failure to supply an opportunistic token to both Samba and Windows clients will result in a termination of the connection.

The authentication process may require numerous round-trips to achieve an authenticated context. During its establishment, the server will respond with a status code indicating that more processing is required alongside the binary blob outputted by `gss_accept_sec_context()`. When it is finalized, the server is required to determine the type of session established: anonymous, guest or authenticated user.

The session type is retrieved by querying the GSS-API using the client context. To check for an anonymous user, the GSS_C_ANON_FLAG is set in the return flags parameter to `gss_accept_sec_context()`. Retrieving the name of a user may be performed by issuing a call to `gss_display_name()`, which in turn may be matched against an implementation-defined guest user name. If these match, the session may be marked as of type guest. The returned flags in SMB2_SESSION_SETUP response, namely SESSION_IS_GUEST and SESSION_IS_NULL, reflect the established session type.

3.3.1 Windows compatibility

There are numerous caveats when facilitating authentication across platforms, even though the specification merely mentions SPNEGO as the only interface to regard. In reality, legacy implementations of the authentication subsystem for Windows require a bit more consideration. Most of these situations arise with NTLMSSP as the chosen GSS authentication mechanic.

Recall from subsubsection 2.3.1, that a client may supply an opportunistic

mechToken sent alongside the list of supported mechanisms in the innerContextToken of SPNEGO. When the supported mechanism is NTLMSSP and an opportunistic mechToken is sent, Windows clients MAY choose to format the token as raw NTLMSSP meant directly for the GSS implementation of NTLMSSP, and not enclosed in an InitialContextToken, which is strictly defined to be *required* by the GSS-API. This behavior is for the sake of backwards compatibility.

When the server supplies an opportunistic list of supported mechanisms in the variable buffer portion of the SMB2_NEGOTIATE response, and this list includes NTLMSSP, a Windows client may choose to ignore SPNEGO all together and directly send a raw NTLMSSP in the following SMB2_SESSION_SETUP, ignoring the mechanism negotiation process. This is also done for the sake of backwards compatibility.

These features, included in the name of backwards compatibility, is known as *Universal Receiver* in the Windows authentication subsystem by the Microsoft documentation [43]. If one wishes to remain inter-operable with clients in the wild, these are issues that needs to be handled.

3.4 Signing messages

SMB facilitates authenticity and integrity guarantees for each exchanged message in an authenticated session. A valid signature lends undeniable evidence that the message was crafted by a known sender, and that it was not altered in transit. The SMB2 header contains a 16 byte field designated to hold a message signature. The generation of this signature differ depending on the negotiated dialect. For versions 2.001 and 2.1 of SMB, the hash based HMACSHA256 is used. For SMB version 3 and later, AES_CMAC-128 is used instead. The signature is generated by inputting the entire request starting at the SMB2 header until the last padding byte sent, including the zero bytes included if part of a compound chain, into the hash function with the session key queried from the authenticated GSS context. This key is retrieved with a single call to `gss_get_mic()` after the session is established. Multiple calls to this function could generate different keys; only the first one is needed. This detail is poorly documented, and you must read the GSS-API RFCs to have the knowledge of its internals to properly grasp this relationship.

The protocol defines a **Security Mode** of operations between the two parties, negotiated through both SMB2_NEGOTIATE and SMB2_SESSION_SETUP. For all dialects of SMB2 this entails three capabilities for signing messages; disabled, enabled or required. If neither server nor client requires signing of messages,

the protocol states they may still sign the message. If either part require signing whilst the other has it disabled, they are incapable of communicating and the connection is terminated.

As discussed in section 3.3, the protocol supports three methods by which a user may establish a session; anonymously, guest, or authenticated user. If the established session is either anonymous or guest, the signing of messages is disabled regardless of the above negotiated *security mode* or if the server has set the **Server→require_message_signing** field. This is a point in the technical document which is left as an exercise to the reader.

4

Design and Implementation

The following chapter explains the architectural principle of our versatile SMB2 server. It highlights the flexibility in expressing implementations of SMB2 commands on a per **Share** basis. Furthermore, it presents some interesting challenges and solutions for the authentication subsystem.

4.1 Design

We have produced a minimal SMB2 server for the Vortex operating system. It supports a functional subset of the features for dialect 2.002. The architectural schema is found in Figure 4.1. The server is layered into separated components, each with its own abstraction and responsibility.

The lower packet handling is concerned with receiving a single message on the transport, as well as shipping a complete response packet. This handling is hidden from the remaining code base, such that any change in transport mechanism does not incur significant changes to the other components. A packet may contain several compounded requests; however, each request is handled individually in its designated handler. Handling compound requests and replying with a compound chain is handled through the packet layer,

alleviating such concerns from the individual request handler.

Each request process the SMB2 header, validating fields and flags to be within protocol specification. After this generic processing, it is shipped to the command handler corresponding to the command identifier. These will in turn perform validation checks universal to the protocol, regardless of the targeted resource. At this point, no actual processing of the requested operation is performed on the server.

The final and most important architectural component is the customizable set of share handlers. Each registered **SHARE** on the server is supplied with a set of functions corresponding to individual commands that may be issued on a given **TREE_CONNECT**. After the generic command processing is finished, the responsibility to complete the operation is yielded to the corresponding command function in the share handler. This is located through **TREE_CONNECT→SHARE**, identified by the supplied **Tree_ID** in the SMB2 header. Commands that do not require a valid **TREE_CONNECT**, such as SMB2_ECHO, do not have entries in the share handler.

SMB exports three different types of shares; disk, IPC, and printer devices. A share handler specify which SMB share type it is, but need not mirror this in the implementation. For instance, a share handler represented as a disk may have nothing to do with the file system. The modularization of commands resembles UNIX virtual file system (VFS), which offers a uniform interface to access the file system regardless of the underlying implementation. The Linux kernel module file system in userspace (FUSE) enables export of the VFS interface to user mode, allowing for alternate behavior through a file system API [1]. Much like VFS and FUSE, the share handlers may implement alternate behavior through the same interface and type.

4.2 Exported shares

We present two share handlers, implemented as SMB disk share types.

The first share handler, vortex_disk, interfaces with the file system interface of Vortex. Each file or directory path is directly mirrored relative to the mount point in the Vortex namespace. Due to a limited feature set corresponding to file and file system operations, some issues arise with this adaption. For instance, Vortex has no support for extended attributes, nor file attributes for

1. Like browsing the front page of the Internet, reddit, through FUSE `https://github.com/redditvfs/redditvfs`

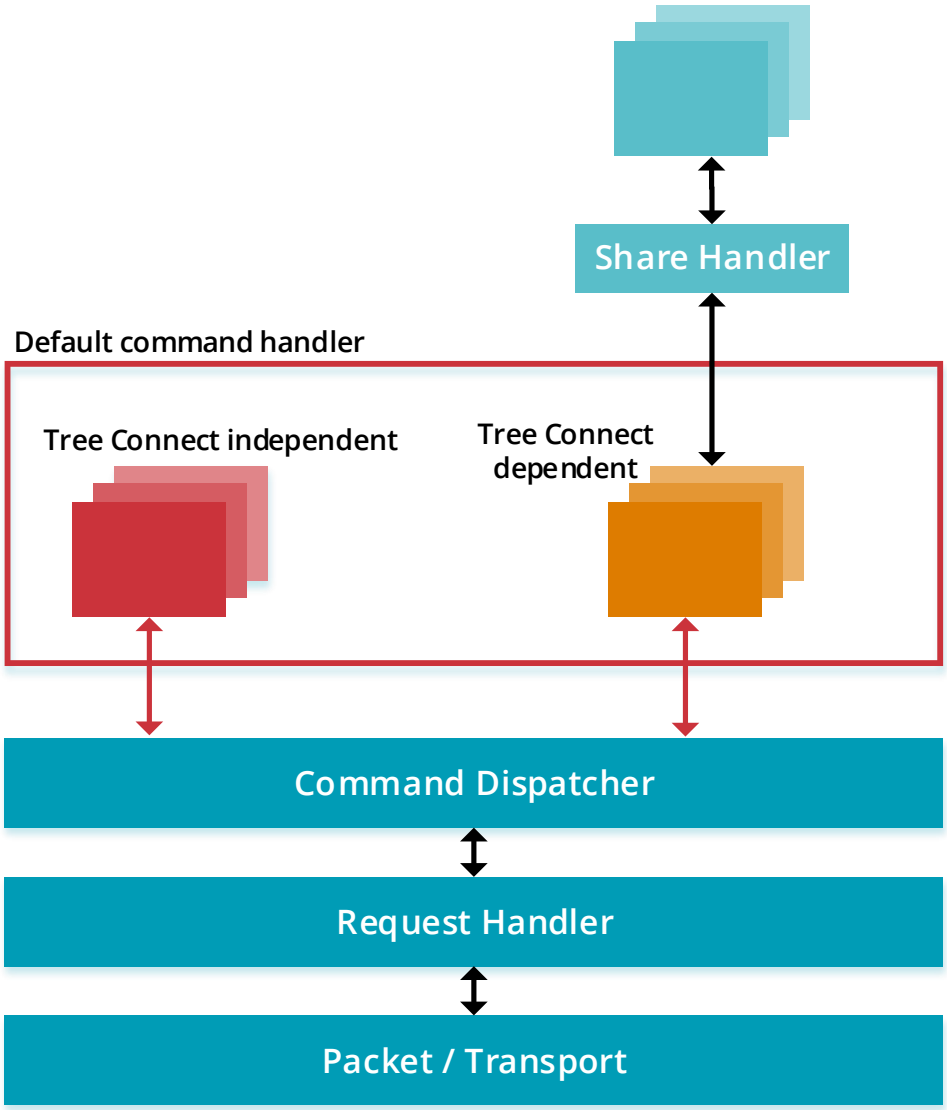


Figure 4.1: The architecture of our SMB2 server. At the bottom layer, retrieving packets from the transport and sending response packets is handled separately from everything else. The packet layer also dispatches each request within the packet to the next layer, the request handler. At this point, generic processing is performed on the request as a whole. Integrity checks, identifier validation and protocol errors are handled before being dispatched to a command specific multiplexer. The command dispatcher identifies which command the current request revolves around, identified by the **Command** in the SMB2 header, and invokes the appropriate command handlers. If the command is related to a TREE_CONNECT, identified by the **TreeID** in the SMB2 header, the execution of the requested command is left to the share handler.

that matter. Nor does it support access control or rights. There is limited notion of file system attributes; querying its capabilities is not exported to userspace. Therefore, populating SMB2_QUERY_INFO commands with sensible data is challenging.

A nontrivial subset of SMB2_QUERY_INFO and SMB2_SET_INFO types are supported by the disk share handler, achieving interoperability with existing clients. Create, read, write, close, and flush are fully supported, enabling file browsing and editing on a mounted disk share. SMB2_CHANGE_NOTIFY is the only command totally lacking support, since Vortex offers no monitoring services. Supporting this command would require a substantial polling framework in our server, with nontrivial requirements.

The second implemented share handler, called vortex_config, exposes the compartment and resource hierarchy in Vortex. It aims to offer configuration and resource monitoring services through a file system interface. It is designed to revolve around a specific compartment, since all operations in Vortex are always performed in the context of one. The root directory reflects the root compartment. Each compartment in turn exports a directory entry for each sub-compartment it contains, named with a `subcompartment_` prefix. Each view of a compartment exports the same set of content.

Additionally, each compartment contains a folder named `processes`. This directory lists a set of files representing running processes for the compartment. Access is restricted so that no read nor write operations may be performed on these files. However, the client has delete access. If a the file is deleted, the corresponding process will be killed on the server.

Similarly, a folder called `start_bin` is exported in each compartment. Within this folder, all access is granted to create and erase files and directories. These files are backed by the in-memory file system exposed by Vortex. There is one reserved filename, `start`. If a SMB2_CREATE request is issued with this name, it is simply denied with an appropriate status code. If a rename operation targets this reserved file name, it will be diverted to a different handler instead of reflecting the rename on the file. The contents of the file is read, expecting a specific configuration format referencing a binary in the file system alongside arguments and environment variables. If the format matches, the referenced binary will be executed in the context of the current compartment.

This configuration share handler implements functionality that is merely toy examples, and may be rather dangerous to export. They are meant to illustrate the versatility of our system, being able to export raw file system operations and complex OS behavior through the same interface.

4.3 Authentication subsystem

In theory, SMB2 supports any authenticating interface that implements GSS-API. However, in practice only two are used. Kerberos 5 authentication provides a mechanism for mutual authentication between a client and server in an open network. It accomplishes this by using symmetric keys to encrypt communication, exchanged through a third-party service known as a key distribution center (KDC). GSS-API bindings are available. NTLMSSP is an authentication protocol used between a client and server deriving a cryptographic key based on a shared secret, typically a password. It is a binary messaging protocol used through Microsoft's security support provider interface facilitating NTLM challenge-response authentication. GSS-API bindings are available.

Given the constrains of working on an experimental operating system, it doesn't have the libraries necessary to SMB2. We are therefore required to introduce this support to Vortex. To reduce the scope of work that is outside of SMB itself, NTLMSSP was the targeted GSS-API mechanism for our authentication stack. Otherwise, communication and facilitation with the third party KDC using Kerberos 5 authentication protocol would be required instead.

As Section 2.3 shows, SMB2 require a total of four libraries to complete the authentication process. Without this support, SMB2 is not able to operate. First of all, the GSS-API library itself, facilitating different mechanisms, is needed. Second, the SPNEGO mechanism is required. It only enables negotiation of mechanisms between the client-server without embedding the specific mechanism within the application. Recall that SPNEGO is the required mechanism for SMB2. Third, a GSS-API mechanism binding with the implementing authentication protocol, NTLMSSP. Forth and last, an actual implementation of NTLM authentication.

4.3.1 Ported libraries

Instead of developing in-house alternatives to the necessary libraries for the authentication subsystem, effort were spent porting existing implementations. Vortex supports a substantial subset of the C standard library, with emulation of many UNIX system calls to achieve binary compatibility with existing executables and object files. The Heimdal suite of authentication libraries is the standard implementation in use on many UNIX distributions, making it an excellent choice. The following libraries are necessary: `libgssapi`, `libgssapi_ntlm`, `libgssapi_spnego`, and `libheimntlm`.

Unfortunately, two of the four libraries rely on UNIX functionality that either had too many levels of nested library dependencies and/or dependencies on

UNIX system calls that are not supported by our emulation. Both of the NTLM libraries were thus ported by removing as many library dependencies as possible. The few pieces of external UNIX libraries they relied on were stripped and ported. In particular, the cryptography primitives were entirely replaced.

The two other GSS-API libraries were compatible out of the box, but came with numerous compiler flags, or lack thereof, that were undesirable for our system. The FreeBSD compile system, from which was the platform the source code was extracted from, makes compiling the required libraries difficult due to hierarchical makefiles and dependencies. Extracting the source into our own repository and build system centralizes dependencies and gives us more control over the build process.

4.3.2 Dynamic loading

Recall from Subsection 2.3.2 that the Heimdal GSS-API facilitation relied on dynamic loading of selected mechanisms, resolved through a configuration file which directs to the location of the shared object file. Unfortunately, a dynamic linker is not natively supported by Vortex. Because of this, the dynamic loader is unavailable, which means that the `dlopen()`, `dlclose()`, and `dlsym()` family of functions will not resolve at linking.

Since the GSS-API library implementation requires the dynamic loader functionality, two options remain: (1) implement a dynamic linker and loader natively in Vortex or (2) provide all the functionality of the dynamic loader without necessarily performing the loading dynamically.

For convenience we chose option (2); the server application was statically linked with all required libraries. Recall from Section 2.2 that the ELF format organizes all symbols and functions, which we can make use of at runtime. The .STRTAB and .SYMTAB sections expose all the information we require, allowing us to lookup a symbol name and locate the function's memory address. Therefore, upon server initialization we load our own binary and parse the content of the ELF header and associated sections. The necessary string and symbol sections are optional for an executable and may be removed at linking or a separate strip operation. Since we require them, the application will terminate with an error condition if no such sections are found, indicating that the server has been erroneously compiled.

Each GSS-API mechanism is expected to implement `_gss_name_prefix()`, which should return a string constant representing the prefix of all GSS functions for the mechanism. This prefix is concatenated with a well-known GSS function name, for example `gss_accept_sec_context()`, and fed into

`dlsym()` to resolve the symbol. The semantics of the operation depend on the mode the shared object file was opened with through `dlopen()`. The symbol resolution would only be matched within the targeted object file, and would not incur a naming clash. In static linking, two equally named symbols supersede each other depending on the linking order. Our solution is thus at odds with the requirements, since the two GSS-API mechanics we wish to employ, SPNEGO and NTLM, both define this prefix resolution symbol but only one is present. This would result in incorrect symbol resolution for one of our mechanics, depending on the order of linking.

Recall the format of the GSS mechanics configuration file listed in Code Snippet 2.3. The shared library column, naming the shared object file which implements the mechanic entry, is the string passed to `dlopen()`. The value of this field is of no significance to us, since we demand all libraries to be statically linked. However, we can leverage this field to supply our own string constant. We implement the C library functions of the dynamic loader, `dlopen()`, `dlsym()`, `dlerr()`, and `dlclose()`. When the dynamic loader is invoked on the string found in the GSS mechanism configuration file, we do not perform any open operation. We merely store this supplied string in the context allocated per `dlopen()`. In a `dlsym()` call, the symbol is first attempted resolved by locating the input name in the associated ELF sections. If no match is found, the name supplied to `dlopen()`, stored in the context, is prepended to the input string and another resolution call attempted. This strategy may accidentally resolve ill-named symbols if there were to exist another symbol equal to the concatenated name, but that is not the case for our libraries. This allows us to list a prefix name in the shared library column, which is dutifully inputted back into `dlopen()` and into our control.

A UNIX utility called *objcopy* is capable of modifying any ELF file. It enables us to redefine symbol names, so that we may prefix the `_gss_name_prefix()` function for both SPNEGO and NTLM GSS-API libraries. Both of these are then prefixed with a string equal to that in the shared library column of the GSS mechanics configuration file. The resolution of `_gss_name_prefix()` is then performed in the context of the library opened upon it, be it NTLM or SPNEGO, yielding the same semantics of a true dynamic loader.

5

Experiments and Evaluation

In this chapter we evaluate the Vortex SMB2 server. First and foremost, we demonstrate its applicability and flexibility. We perform throughput testing and comparison against a contemporary server for the Ubuntu operating system.

5.1 Mounted shares

We have successfully developed two share handlers and exposed them as shares on our server.

The first implements a disk share type, exposing the full file system of Vortex. A screenshot of an Ubuntu client browsing the mounted file system share is shown in Figure 5.1. It supports a sufficient subset of commands to interact with the file system as you would if it resided locally. Streaming of video, document editing, file renames, deletion, etc., all work as expected.

The second share handle illustrates the versatile design of our server, exposing the administrative interface with the resources described in Section 4.2. Figure 5.2 shows two screendumps of navigation within the configuration share

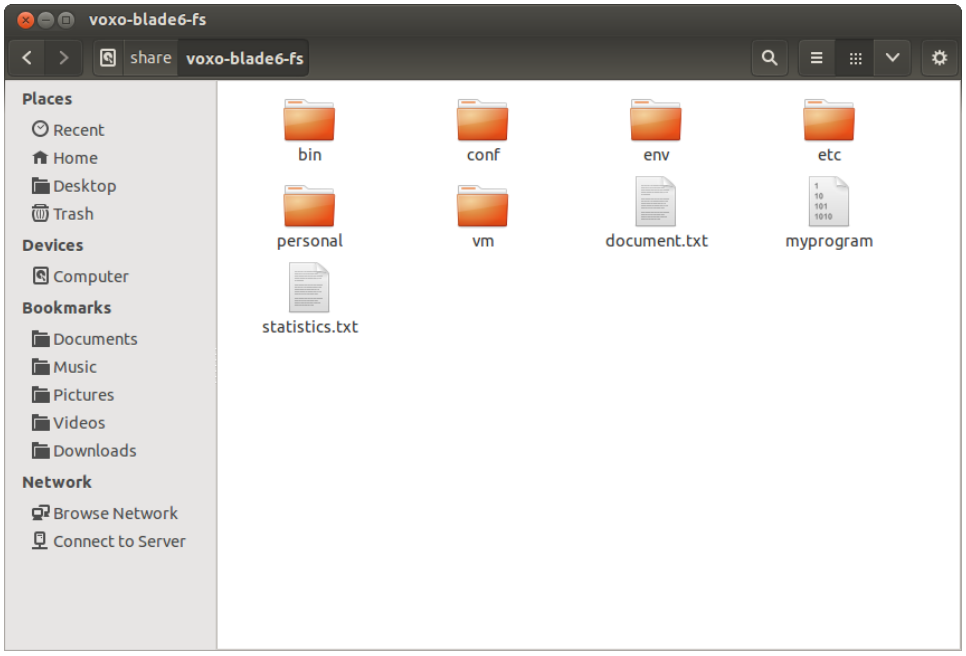


Figure 5.1: Screenshot of an Ubuntu desktop, accessing a disk share on the Vortex file system through its native file browser.

opened in a file browser on an Ubuntu desktop. The top screenshot is of the root compartment hierarchy, exposing a set of sub-compartments, a directory containing all running processes in the current compartment, and a specialized directory containing text files that can be used to start new binaries. The processes directory contains a set of files, shown in the bottom file browser, representing running processes which are named after the their process binary and process id. Deleting a file maps to killing the corresponding process. Reading and writing to these files is prohibited through access restrictions.

5.2 Throughput performance comparison

We run our experiments on Hewlet Packard ProLiant BL460c G1 blade servers. These are equipped with twin Intel Xeon 5355 processors running at a peak frequency of 2.66 GHz, with 16GB of pc2-5300 DDR2 RAM. The server blades are connected to a 1 Gb/s Ethernet network. We perform our experiments from an Ubuntu 14.04 desktop which acts as the SMB2 client. Throughput is measured by the amount of load on the ETH0 interface during the experiment, and retrieved through the `ifstat` Linux utility.

We performed copying of large files from the local file system to a mounted par-

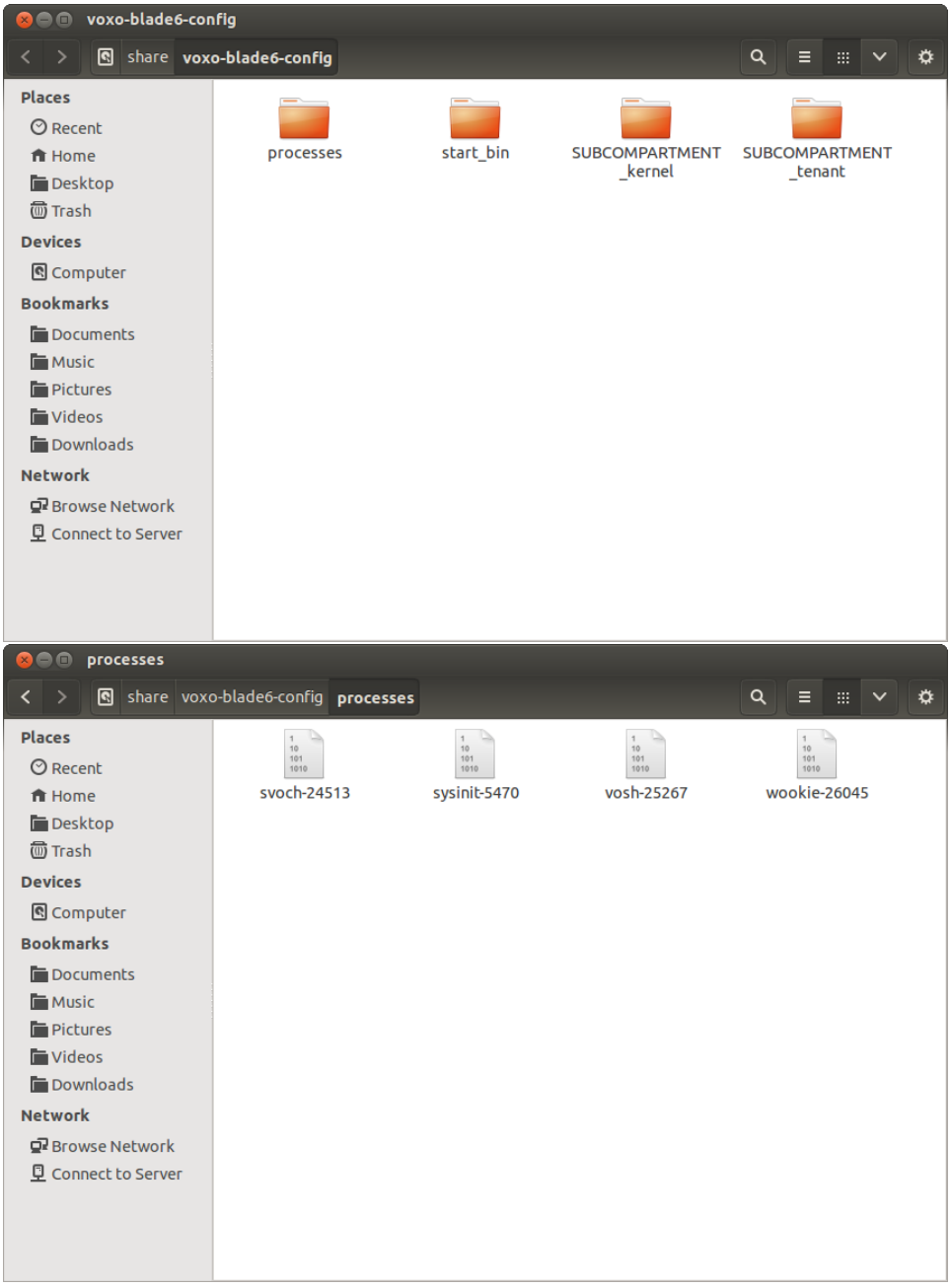


Figure 5.2: Two screenshots of an Ubuntu file browser with two different working directories. The top view displays the root configuration share, exposing four directories. The two directories prefixed with SUBCOMPARTMENT_ represents child compartments of the root. The contents of the processes directory is shown in the bottom file browser. It enumerates all the running processes in the root compartment.

tition. Its worth noting that several utilities are available to perform this copying operation in Linux; `cat`, `cp`, `mv`, `scp`, or `rsync`. We perform our experiments with `rsync`.

Figure 5.3 plots the maximum throughput achieved with both reading and writing to and from Vortex over a TCP stream. The content transferred is discarded so that we eliminate any processing time on the content. As evident from the graph, the throughput per second for both read and especially write requests are highly varied. This spiking in performance is hard to pinpoint, but scheduler interference, memory access patterns, or interrupt handlers taking up varied CPU time are valid assumptions. The possible issues related to raw TCP throughput in Vortex is however not within the scope of our work. We concern ourselves with the performance of our SMB2 server over a TCP stream, measuring and comparing relative to our baseline measurements.

SMB2 enables congestion control in the form of outstanding credits, as described in Subsection 3.2.2. If the client is granted multiple outstanding credits, it may dispatch several operations concurrently. Neither requests nor responses are required to be sent or received in-order of their **Message_id**, only that they are within bounds of the outstanding credit score. This opens up the possibility to handle incoming requests and dispatching a response in separate threads.

Throughput performance may vary greatly depending upon the copying utility. For instance, the block size used in the copying operation may not saturate the byte transfer capacity of the SMB2 operation. The maximum PDU size of each operation in SMB2 are constrained by negotiating **Connection→max_read_size**, **Connection→max_write_size**, and **Connection→max_transaction_size**. For SMB2 dialect 2.002, the maximum value to these fields are 2^{16} bytes. If the block size of the copying utility issues operations larger than this negotiated maximum, either the operating system or the SMB2 client must split and buffer the block into several SMB operations.

We measure the read throughput by retrieving a 3 gigabyte file from the server over a disk file system share. Figure 5.4 shows a comparison between our Vortex server and a contemporary Ubuntu server. We achieve an average throughput of roughly 45 MB/s, while Samba achieves an average of 61 MB/s on Ubuntu.

We measure the write throughout by transferring a 3 gigabyte file to the server over a disk file system share. Figure 5.5 plots the throughout achieved by our Vortex server and the Samba server running on Ubuntu. We achieve an average throughout of 54 MB/s, while Samba averages 63 MB/s on Ubuntu.

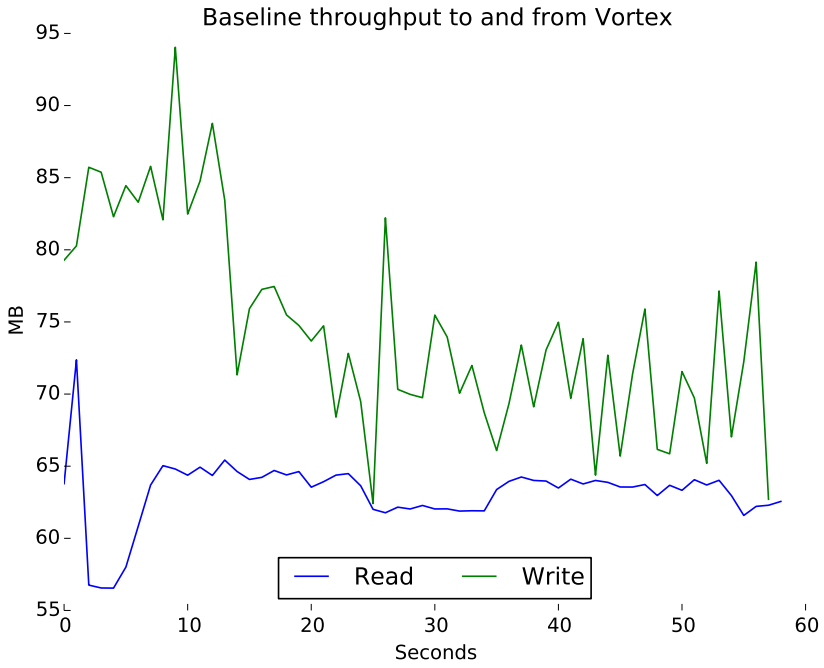


Figure 5.3: A maximum throughput test to and from Vortex. A simple server on Vortex accepted clients, reading and writing fixed size null data that was simply ignored. A similar client application was used from an Ubuntu workstation, over a 1 Gb/s Ethernet network link.

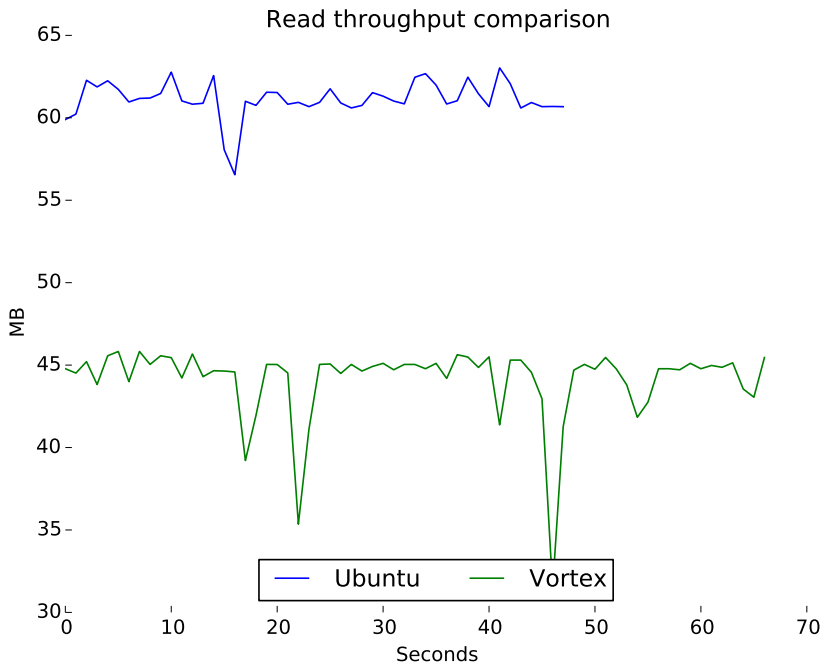


Figure 5.4: A throughput comparison of reading a 3GB file from a remote disk share to an Ubuntu client over a 1 Gb/s Ethernet network. The Vortex server is on average 15 MB/s slower than its Ubuntu counterpart.

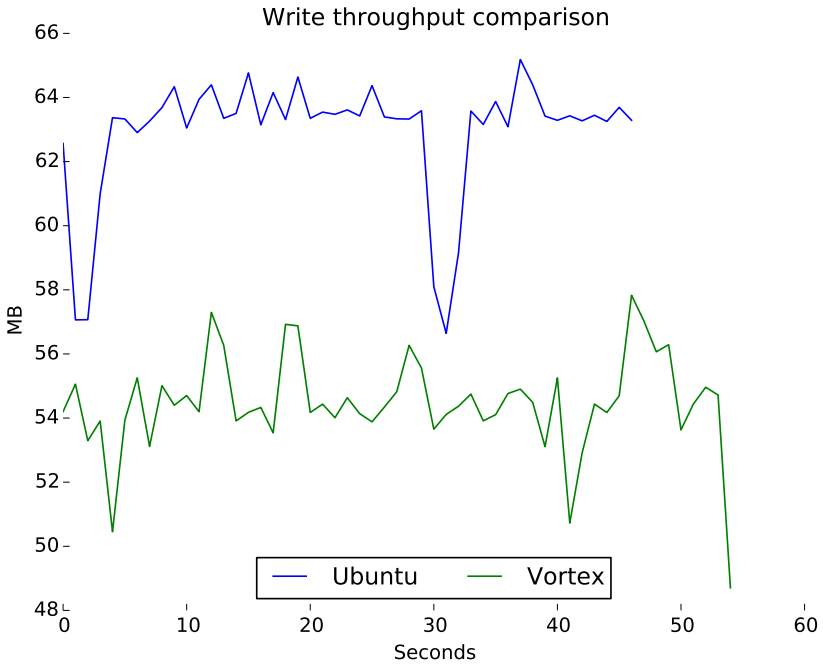


Figure 5.5: A throughput comparison of writing a 3 GB file from an Ubuntu client to a remote share over a 1 Gb/s Ethernet network. The Vortex server is on average 10 MB/s slower than its Ubuntu counterpart.

We do not achieve the same average transfer speed as our Ubuntu counterpart. When we compare the throughput achieved by Ubuntu to our maximum transfer speed over TCP, there is no room left for processing time. Therefore, to achieve the same or greater throughput as Samba on Ubuntu, it would require revisiting the implementation of the entire network and I/O stack in Vortex.

Our current implementation is single-threaded.[1] That is, only one thread is active in processing packets from any client. Inspecting the packets sent on the wire during a `rsync` file copying operation revealed that four SMB2_READ or SMB2_WRITE requests were batched at any time, with the maximum transaction size possible by the current dialect. Efforts to parallelize request and response handling on the same object led to severe throughput reduction. We attribute this to lock contention and/or reduced cache efficiency on the I/O aggregate for the same file object between multiple competing threads. A key aspect of getting greater performance may be to efficiently parallelize request handling, which is left to future work.

We have produced an SMB2 server with a single-thread architecture. We achieve acceptable throughput performance, albeit not as high as Samba running on Ubuntu 14.04.

1. The Samba implementation is also single-thread. It will `fork()` after a new connection is accepted.

6

Concluding Remarks

In this chapter, we elaborate areas of future work to improve both performance and interoperability of our SMB2 server. We then conclude the thesis.

6.1 Future work

Although we achieve full interactive access to the file system with existing clients, many niche features are left unimplemented. Commands such as SMB2_QUERY_INFO and SMB2_IOCTL presents a myriad of sub-operations, many of which simply return a not supported status code.

The Vortex file and file system abstractions are minimal, completely lacking common attributes exchanged by the protocol. At present, responses to commands operating upon such fields return a default emulation of the requested operation. However, the core Vortex abstractions should be extended to provide more fine grained control of file resources. An example of an operation which cannot be easily supported without modification to the native file interface is the SMB2_SET_INFO **end_of_file** operation. The command references a given file, giving it a new size regardless of the old. If this new size is greater than the current file size, the file is simply zero-extended. However, Vortex only supports truncating a file down to zero bytes. This renders us unable to fulfill the semantics of the requested operation natively. To achieve this, complete file copies must be performed to emulate the behavior.

Finally, implementing more feature rich dialects for SMB2 would enable greater performance and bring encryption of entire packets allowing confidentiality between communicating parties.

6.1.1 Pipe share type

At present, there is no support for the pipe share type. There exists a default pipe share named **$IPC** which as the name implies, performs inter-process communication. The data transfered through SMB2_READ and SMB2_WRITE operations are designed for a service, named by the SMB2_CREATE command, which communicates through the Distributed Computing Environment / Remote Procedure Calls (DCE/RPC) protocol. This system is needlessly complex, designed by a committee on commission from the Open Software Foundation. The DCE/RPC framework makes no assumptions about the underlying transport, embedding features such as packet fragmenting and reconstruction, signing, and checksumming into its protocol. Many of these features are superfluous over an established SMB session, since its transport takes care of these concerns. The only fathomable reason to use DCE/RPC to target these services, is that they should be reachable through direct means which requires the features otherwise provided by the SMB transport and session.

Providing support for the DCE/RPC subsystem would allow the creation of a **$IPC** share to handle the requested RPC services. Windows SMB clients insists on connecting to the **$IPC** share and opening several services across it. Regardless of the status code returned, they will simply retry the operation. This has the undesirable side-effect of making all other operations with Windows clients impossible, effectively rendering our server unable to function properly with these. Facilitating the DCE/RPC subsystem would enable interoperability.

6.1.2 Context-sensitive share handlers

We have successfully demonstrated a versatile architecture, enabling us to construct context-sensitive share handlers for command operations. There exists great potential in the resource management capabilities exposed as part of our configuration share. For example: adjusting resources, constructing new, or deleting old compartments.

Another area that could make use of the context-sensitive share handlers is exposing a read-only interface to server statistics. Currently, this information is only available through a specialized format read over a raw TCP connection. Exposing this information in a well-defined file system hierarchy enables an

external application to easier interface and utilize this data.

6.2 Concluding remarks

We have successfully deployed a minimal SMB2, dialect 2.002 server to Vortex. It follows a versatile design and implementation, allowing us to build context-sensitive handling of commands on a per share basis. Users of the Vortex operating system are now able to browse, edit, copy and retrieve file objects residing on a remote node, from the comforts of their own desktops through contemporary clients. Its extensibility enables remote resource management through the conventional file system interface.

Bibliography

[1] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks". *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[2] Åge Kvalnes, "Omni-Kernel: An Operating System Architecture for Pervasive Monitoring and Scheduling". Ph.D. dissertation, UiT The Arctic University of Norway, 2013.

[3] A. Kvalnes, D. Johansen, R. van Renesse, S. Valvag, and F. Schneider, "Omni-Kernel: An Operating System Architecture for Pervasive Monitoring and Scheduling". *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, p. 1, 2015.

[4] A. Nordal, A. Kvalnes, J. Hurley, and D. Johansen, "Balava: Federating private and public clouds", in *Services (SERVICES), 2011 IEEE World Congress on*, IEEE, pp. 569–577, 2011.

[5] A. Nordal, Å. Kvalnes, and D. Johansen, "Paravirtualizing tcp", in *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, ACM, pp. 3–10, 2012.

[6] A. Ø. Nordal, Å. Kvalnes, R. Pettersen, and D. Johansen, "Streaming as a hypervisor service", in *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, ACM, pp. 33–40, 2013.

[7] J.-O. Karlberg, "ROPE: Reducing the Omni-kernel Power Expenses". Master's thesis, UiT The Arctic University of Norway, 2014.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization". *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", in *Proceedings of the Nineteenth ACM Symposium on Operating Systems*

Principles, pp. 29–43, 2003.

[10] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency", in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, pp. 143–157, 2011.

[11] "Server Message Block (SMB) Protocol Version 2 and 3". Microsoft Corporation, [STANDARD], MS-SMB2, Rev. v20140502, May 2014.

[12] IBM, "Announcement Letter Number 184-100". [ONLINE], Available: http://www-01.ibm.com/common/ssi/printableversion.wss?docURL=/common/ssi/rep_ca/0/897/ENUS184-100/index.html. Retrieved: 20.03.2015, IBM PC Network Announcement Letter, dated August 14 1984.

[13] OSI, "Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model". OSI/IEC, [STANDARD], OSI/EIC 7498-1:1994(E), November 1994, Second edition. First edition published in 1984.

[14] IBM, Tech. Rep. "LAN Technical Reference: 802.2 and NetBIOS APIs".

[15] IEEE, "Logical Link Control". IEEE Computer Society, [STANDARD], IEEE 802.2-1985, 1984.

[16] IEEE, "Standards for Local Area Networks: Token Ring Access Method and Physical Layer Specifications". IEEE Computer Society, [STANDARD], IEEE 802.5-2:1989, 1989.

[17] T. D. Evans, "NetBIOS, NetBEUI, NBF, NBT, NBIPX, SMB, CIFS Networking". 1998.

[18] N. W. G. in the Defense Advanced Research Projects Agency, I. A. Board, and E. to End Services Task Force, "Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods". RFC 1001 (INTERNET STANDARD), Available: http://www.ietf.org/rfc/rfc1001.txt. Mar. 1987.

[19] N. W. G. in the Defense Advanced Research Projects Agency, I. A. Board,

and E. to End Services Task Force, “Protocol standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications”. RFC 1002 (INTERNET STANDARD), Available: http://www.ietf.org/rfc/rfc1002.txt. Mar. 1987.

[20] “System V Application Binary Interface”. AT&T Unix System Laboratories, Specification, Release 4, 1988.

[21] “Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2”. TIS Committee, Specification, 1995.

[22] IETF, “Common Authentication Technology (cat)”. [ONLINE], Available: https://datatracker.ietf.org/wg/cat. Retrieved: 03.04.2015.

[23] J. Linn, “Common Authentication Technology Overview”. RFC 1511 (Informational), Available: http://www.ietf.org/rfc/rfc1511.txt. Sept. 1993.

[24] J. Linn, “Generic Security Service Application Program Interface”. RFC 1508 (Proposed Standard), Available: http://www.ietf.org/rfc/rfc1508.txt. Sept. 1993, Obsoleted by RFC 2078.

[25] J. Wray, “Generic Security Service API : C-bindings”. RFC 1509 (Proposed Standard), Available: http://www.ietf.org/rfc/rfc1509.txt. Sept. 1993, Obsoleted by RFC 2744.

[26] J. Linn, “Generic Security Service Application Program Interface Version 2, Update 1”. RFC 2743 (Proposed Standard), Available: http://www.ietf.org/rfc/rfc2743.txt. Jan. 2000, Updated by RFC 5554.

[27] J. Wray, “Generic Security Service API Version 2 : C-bindings”. RFC 2744 (Proposed Standard), Available: http://www.ietf.org/rfc/rfc2744.txt. Jan. 2000.

[28] E. Baize and D. Pinkas, “The Simple and Protected GSS-API Negotiation Mechanism”. RFC 2478 (Proposed Standard), Available: http://www.ietf.org/rfc/rfc2478.txt. Dec. 1998, Obsoleted by RFC 4178.

[29] L. Zhu, P. Leach, K. Jaganathan, and W. Ingersoll, “The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism”. RFC 4178 (Proposed Standard), Available: http://www.ietf.org/rfc/rfc4178.txt. Oct. 2005.

[30] “Server Message Block (SMB) Protocol”. Microsoft Corporation, [STAN-

DARD], MS-SMB, Rev. v20140502, May 2014.

[31] Microsoft, "Common Internet File System (CIFS) Protocol". Microsoft Corporation, [STANDARD], MS-CIFS, Rev. v20140502, May 2014, First edition released in 2009.

[32] I. Heizer, P. Leach, and D. Perry, "Common Internet File System Protocol (CIFS/1.0)". IETF Secretariat, [DRAFT], June 1996, draft-heizer-cifs-v1-spec-00.txt.

[33] C. R. Hertel, "Implementing CIFS: The Common Internet File System", 1st ed. Prentice Hall Professional, 2004.

[34] N. Virk and P. Prahalad, "What's new in SMB in Windows Vista". [ONLINE], Available: http://blogs.msdn.com/b/chkdsk/archive/2006/03/10/548787.aspx. Retrieved: 23.05.2015.

[35] J. Barreto, "SMB2, a complete redesign of the main remote file protocol for Windows". [ONLINE], Available: http://blogs.technet.com/b/josebda/archive/2008/12/05/smb2-a-complete-redesign-of-the-main-remote-file-protocol-for-windows.aspx. Retrieved: 25.05.2015.

[36] "SMB2 Remote Direct Memory Access (RDMA) Transport Protocol". Microsoft Corporation, [STANDARD], MS-SMBD, Rev. v20140502, May 2014.

[37] "Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1 Annex A17: RoCEv2". InfiniBan Trade Association, [STANDARD], Annex A17: RoCEv2, September 2014.

[38] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A Remote Direct Memory Access Protocol Specification". RFC 5040 (Proposed Standard), Available: http://www.ietf.org/rfc/rfc5040.txt. Oct. 2007.

[39] H. Shah, J. Pinkerton, R. Recio, and P. Culley, "Direct Data Placement over Reliable Transports". RFC 5041 (Proposed Standard), Available: http://www.ietf.org/rfc/rfc5041.txt. Oct. 2007.

[40] "InfiniBand Architecture Specification Volume 1 Release 1.2.1". InfiniBan Trade Association, [STANDARD], Release 1.2.1, November 2007.

[41] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels". RFC 2119 (Best Current Practice), Available: http://www.ietf.org/rfc/rfc2119.txt. Mar. 1997.

[42] “Windows Error Codes”. Microsoft Corporation, [STANDARD], MS-ERREF, Rev. v20140502, May 2014.

[43] “Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) Extension”. Microsoft Corporation, [STANDARD], MS-SPNG, Rev. v20140502, May 2014.