NFSv4 D. Noveck
Internet-Draft NetApp
Updates: 8881, 7530 (if approved) 16 May 2025
Intended status: Standards Track
Expires: 17 November 2025

Internationalization for the NFSv4 Protocols
draft-ietf-nfsv4-internationalization-12

Abstract

This document describes the handling of internationalization for all NFSv4 protocols, including NFSv4.0, NFSv4.1, NFSv4.2 and extensions thereof, and future minor versions.

It updates RFC7530 and RFC8881.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 17 November 2025.

Table of Contents

1. Introduction

Internationalization is a complex topic with its own set of terminology (see [RFC6365]). The topic is made more difficult to understand for the NFSv4 protocols by the complicated history described in Appendix C. In large part, this document is based on the actual behavior of NFSv4 client and server implementations (for all existing minor versions). It is intended to serve as a basis for further implementations to be developed that can interact with existing implementations. It is expected to enable interoperation with implementations to be developed in the future.

Note that the set of behaviors on which this document is based are each effected by a combination of an NFSv4 server implementation proper and a server-side underlying file system. It is common for servers and underlying file systems to be configurable as to the behavior shown. In the discussion below, each configuration that shows different behavior is to be considered separately.

As a consequence of this approach, normative terms defined in [RFC2119] are often derived from implementation behavior, rather than the other way around, as is more commonly the case. The specifics are discussed in Section 2.

With regard to the question of interoperability with existing specifications for NFSv4 minor versions, different minor versions pose different issues, even though the actual behavior is the same for all minor versions. This is because some of the specifications were often adopted without the appropriate concern for usability, implementability, or the expectations of existing NFS users.

* With regard to NFSv4.0 as defined in [RFC7530], no significant interoperability issues are expected to arise because the discussion of internationalization in that specification, which is the basis for this one, was also based on the behavior of existing implementations. Although, in a formal sense, the treatment of internationalization here supersedes that in [RFC7530], the treatments are intended to be the same, in order to eliminate the possibility of interoperability issues.

Because of a change in the handling of Internationalized domain names, there are some differences from the handling in [RFC7530], as discussed in Appendix C. For a discussion of those differences and potential compatibility issues, see Sections 11.1 and 11.2.

* With regard to NFSv4.1 as defined by [RFC8881], the situation is quite different. The approach to internationalization specified in that document, based in large part on that in RFC3530, was never implemented, and implementers were either unaware of the troublesome implications of that approach or chose to ignore the existing specifications as essentially unimplementable. An internationalization approach compatible with that specified in [RFC7530] tended to be followed, despite the fact that, in other respects, NFSv4.1 was considered to be a separate protocol from NFSv4.0.

  If there were NFSv4 servers who obeyed the internationalization dictates within [RFC5661], or clients that expected servers to do so, they would fail to interoperate with typical clients and servers when dealing with non-UTF8 file names, which are quite common. As no such implementations have come to our attention, it has to be assumed that they do not exist and interoperability with existing implementations as described here is an appropriate basis for this document.

  The same applies to all existing minor versions beyond NFSv4.1 (i.e. to NFSv4.2), which made no changes in the specification of internationalization-related handling and for which existing implementation patterns were maintained.

There is one area within the protocol for which existing implementations are somewhat limited, so that it is not always possible to derive the details of the specification from existing implementations. This area addresses situations in which, in response to user needs, it is necessary to treat distinct strings as equivalent based on an equivalence relation applying to UTF8-encoded Unicode strings. In order to provide this internationalization-related functionality, it is necessary, as described in Section 6, for the server to be aware of the encoding of strings used for file names, as UTF8-encoded Unicode.

There are several classes of equivalence relations, for which we have limited implementation experience:

* NFSv4 implementations MAY treat two canonically equivalent strings as denoting the same object.

While the ability for servers to do that is an NFSv4 design requirement necessary to provide support for Unicode normalization, and some implementations do exist, there has, so far, been little demand for this feature and current implementations are not heavily used.

As a result, the support for such features described here, while derived from implementation experience, has only been used in a small set of situations and might have difficulties with some existing clients that do various forms of name caching. See Section 7.1 for further discussion.

* NFSv4 implementations MAY treat two strings that differ only as to case as denoting the same object. While server implementations exist, the details are unclear because of the complexity of case-mapping and case-based string equivalence in an internationalized environment.

  Because the details of case mapping and case-insensitive string comparison can be complex in an internationalized environment, with desirable mappings depending on user preference and the use of different languages, the definition of appropriate mappings cannot be done within this specification, although the issues that need to be dealt with are discussed in Section 7.2

2. Terminology

2.1. Requirements Language Definition

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2.2. General Definitions

The following terms are used in this document as defined below.

Canonical Equivalence (of strings): In Unicode, two strings are considered canonically equivalent if they can be assumed to have the same appearance and meaning when printed or displayed.

For further detail and examples, see Section 7.1.

Case-insensitive File System treat file names that differ only in case (e.g. "a" and "A") as the same, allowing only one such to exist in a given directory.

The decision as to whether two strings differ only as to case can be a complicated one in general, because different languages have different rules (e.g. dotted and dotless i’s in Turkic languages) and because different versions of Unicode include different sets of characters with different case mappings.

Case-sensitive File System treat file names that differ only in case (e.g. "a" and "A") as distinct, allowing each to designate a different file in a directory.

Such file systems are easier to deal with because they do not to define case mappings and are consistent with the assumptions of POSIX.

Underlying File System The realization of a server-side file systems used to implement requests made using the NFSv4 protocol.

Most often, such file systems can be used by other remote access protocols or to effect locally requested file operations

UTF8-aware File System assume use of Unicode as encoded using UTF-8 by both client and server.

This shared knowledge allows the server to support case-insensitive file systems and those that treat canonically equivalent names as designating the same file.

UTF8-unaware File System do not make any assumptions as to the interpretation of the strings within component names.

Two component names are considered equivalent only if they are identical.

Such file systems cannot be case-insensitive or deal with Unicode normalization issues.

3. Internationalization and Minor Versioning

Despite the fact that NFSv4.0 and subsequent minor versions have differed in many ways, the actual implementations of internationalization have remained the same and internationalized file names have been handled without regard to the minor version being used. Minor version specification documents contained different treatments of internationalization as described in Appendix C but of those only the implementation-based approach used by [RFC7530], resulted in a workable description while a number of attempts to specify another approach that implementers were to follow were all ignored by implementers.

It is expected that any future minor versions will follow a similar approach, even though it is possible that a future minor version will adopt a different approach as long as the rules within [RFC8178]) are adhered to. In any such case, the new minor version would have to be marked as updating or obsoleting this document. Some issues relating to potential extensions within the framework specified in this document are dealt with in Appendices A.3 and A.4.

4. Changes Relative to RFC7530

This document follows the internationalization approach defined in RFC7530, with a number of significant changes listed below, all necessary to provide an updated treatment that can be used for all minor versions.

The making this shift, the handling of internationalization specified in [RFC7530] is applied to all NFSv4 minor versions. No compatibility issues are expected to arise because all existing implementations follow the same approach to internationalization despite the large difference between [RFC7530] and what is specified in [RFC8881].

The following changes were necessary:

* Issues relating to potential future minor versions and protocol extensions are addressed in Appendix D.

* Changes made necessary by the shift from IDNA2003 to IDNA2008 have been made. The intention is to maintain compatibility with all existing implementations of all NFSv4 minor versions. Potential compatibility issues with regard to the IDNA shift are discussed in Section 11.2.

* There is more discussion of case-insensitive handling of file names, with particular attention to the complexities that can arise when multiple language conventions in these matters need to be accommodated. Because of the need to accommodate these complexities, the protocol leaves these details up to the server while the material in Appendices A.1 and A.2 provides a helpful introduction to these issues.

* There is additional material, dealing with the implications of server-side internationalization-related file name processing for clients' use of certain name caching techniques. This includes a discussion of options to deal with the current lack of detailed information about the server (in Sections 7.1 and 7.2, and options for handling this issue until more detailed information can be made available to the client (in Section 7.3)."

* A discussion of the OPTIONAL attribute fs_charset_cap has been added.

* A previous discussion of the behavior of certain file systems that could be construed as suggesting (even though the words "SHOULD NOT were used, that it was valid for a server to perform normalization-related processing on names without rejecting names that are not valid UTF-8 strings.

  That text has now been deleted and other text clarifies that this is not valid behavior.

5. Limitations on Internationalization-Related Processing in the NFSv4 Context

There are a number of noteworthy circumstances that limit the degree to which internationalization-related encoding and normalization-related restrictions can be made universal with regard to NFSv4 clients and servers:

* The NFSv4 client is part of an extensive set of client-side software components whose design and internal interfaces are not within the IETF's purview, limiting the degree to which a particular character encoding might be made standard.

* Server-side handling of file component names is most often implemented within a server-side underlying file system, whose handling of character encoding and normalization is not specifiable by the IETF.

* Typical implementation patterns in UNIX systems and the POSIX handling of file name strings result in the NFSv4 client having no knowledge of the character encoding being used, which might even vary between processes on the same client system.

* Users may need access to files stored previously with non-UTF-8 encodings, or with UTF-8 encodings that are not in accord with any particular normalization form.

Despite the above, there are cases in which UTF8-related processing can be provided by servers, as described in Sections 7 and 6.

6. Server Behavior Types

There are two basic types of server filesystems supported by NFSv4, which differ in their handling of internationalization- related issues, as they apply to the handling of the names of file system objects. The details of how these types affect the handling of potential string equivalence relationships are discussed in Section 7.

These two types of file systems can be distinguished based on the value of the flag FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 in the value returned by the fs_charset_cap attribute.

* Servers which do not rely on knowledge of the encoding used for name strings are termed "UTF8-unaware". Because such servers, when handling file names, do not rely on any particular encoding being used, they can be used with a range of character encodings, in the same way that was done when using NFSv3.

  This flexibility is necessary to enable access to existing files stored with names using existing encodings. However, the lack of server knowledge of the encoding used results in such servers' inability to provide the kind of services described in Section 7 that rely on the ability to treat sets of distinct strings as equivalent, for the purpose of handling normalization issues and providing case-insensitivity.

  Because the server has no ability to define name string equivalence relations, clients can cache names without knowledge of the encoding used by the server.

* Servers that are aware of the encoding of strings using the UTF-8 encoding of Unicode are termed "UTF8-aware". Such servers are able to provide normalization-related handling as described in Section 7.1 and case-insensitivity as described in Section 7.2 by defining equivalence relations that treat defined sets of strings as equivalent for naming purposes.

  Because of the ability of such servers to define name equivalence relations, certain forms of name caching can be interfered with because the client is not aware of the equivalence relation used. Because of this lack of knowledge, forms of name caching where the name used to refer to a file is not expected to change can be interfered with.

In the case of UTF8-aware filesystems, server decisions with regard to normalization handling and case-insensitivity are independent but implementers need to be aware of some potential interactions.

* Because there is no way for the client to determine whether normalization-related processing is in effect, the client might need to act as if it is used for all UTF8-aware file systems.

* When both normalization-related processing and case-insensitivity are to be implemented, those two functions can be provided together. The server can use string equivalence relations that provide both functions, by treating two strings as equivalent if they are canonically equivalent or differ only as to case.

  See Appendix B.2 for a discussion of implementing string comparisons given the existence of such a common equivalence. It is worth nothing that, when clients are made aware of server string equivalence relations, using facilities such as those described in Appendices A.3 and A.4, the client and server can use the same string equivalence relation, enabling the previously necessary restrictions on client-side name caching to be eliminated.

7. Handling of String Equivalence

Although many NFSv4 implementations continue the approach to string names used in NFSv3 in which the only equivalent strings are identical, others provide support for various sort of string equivalence relations as described in Sections 7.1 and 7.2 below.

The earlier approach dealt with internationalization outside the scope of the protocol, by making internationalization the job of the user, requiring the client user and server to agree on the character encoding being used while the implementations themselves strived for character-encoding neutrality with knowledge of the encoding by the implementations limited to the encoding of strings such as "/", ".", and "..".

As discussed later in Section 6, NFSv4 supports multiple modes of operation in dealing with these matters. While NFSv4 supports the older mode of operation by allowing UTF8-unaware file systems, the protocol also supports the use of UTF8-aware file systems in which both sides of the implementation deal with filenames as UTF8-encoded Unicode strings, enabling equivalence classes of those strings to be used within the protocol.

When equivalence classes of string are implemented, this can be done in two ways:

* Equivalent strings are treated as identical in matching names with associated files. This typically requires special code within the server-side file system, rather than in the server proper.

* Name strings may be mapped to equivalent names resulting in a file having an equivalent name rather than the one specified by the client. This approach is implementable within the server proper.

The existence of distinct equivalent strings does not, by and large, cause troublesome issues for clients, who can function without detailed knowledge of the equivalence relation(s) implemented. However, as noted in Section 7.3, certain forms of client caching are not workable or need to be heavily restricted, in environments in which such string equivalences re implemented by the server.

7.1. Handling of Canonical Equivalence of Strings

It is often desirable to treat two strings that are essentially the name, although normalized differently, as equivalent. Such equivalences can arise in multiple ways:

* In some cases, two Unicode values are assigned to a single glyph, because those two values represent different meanings of the same symbol. For example, OHM SIGN (U+2126) denotes the same symbol as GREEK CAPITAL LETTER OMEGA (U+03A9) and the two are considered canonically equivalent.

* There are a large number of situations in which a particular symbol can be represented as a single character or as a combination of a base character and a combining character adding a diacritic. For example, LATIN CAPITAL LETTER E ACUTE (U+00C9) can also represented by LATIN CAPITAL LETTER E (U+0045) followed by COMBINING ACTUE ACCENT (U+0301). These two strings are canonically equivalent.

  Generally, when such pairs exist, the form in which the diacritic is integrated into the symbol is designated the NFC form while the other is the NFD form.

Whenever a set of at least two canonically equivalent strings exists, one of these is one that is the NFC form and one is the NFD form. These are usually different although this is not always the case. Some examples:

1. OHM SIGN (U+2126) is canonically equivalent to GREEK CAPITAL LETTER OMEGA (U+03A9).

   In this case, the NFC and NFD forms are the same and both are GREEK CAPITAL LETTER OMEGA (U+03A9).

2. The two strings LATIN CAPITAL LETTER E ACUTE (U+00C9) and LATIN CAPITAL LETTER E (U+0045) followed by COMBINING ACTUE ACCENT (U+0301) are canonically equivalent.

   In this case, the NFC form is LATIN CAPITAL LETTER E ACUTE (U+00C9) while the NFD form is LATIN CAPITAL LETTER E (U+0045) followed by COMBINING ACTUE ACCENT (U+0301).

3. The three strings ANGSTROM SIGN (U+212B), LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5), and LATIN CAPITAL LETTER A (U+0041) followed by COMBINING RING ABOVE (U+030A) are all canonically equivalent

   In this case, the NFC form is LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5) while the NFD form is LATIN CAPITAL LETTER A (U+0041) followed by COMBINING RING ABOVE (U+030A).

4. Sets of canonically equivalent strings can be arbitrarily large. For example, the twelve strings each consisting of one string from each of 1), 2), and 3) above are all canonically equivalent.

   In this case, the NFC form is of each of these twelve strings GREEK CAPITAL LETTER OMEGA (U+03A9) followed by LATIN CAPITAL LETTER E ACUTE (U+00C9) followed by LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5).

   In contrast, the NFD form of each of these twelve strings is GREEK CAPITAL LETTER OMEGA (U+03A9) followed by LATIN CAPITAL LETTER E (U+0045) followed by COMBINING ACTUE ACCENT (U+0301) followed by LATIN CAPITAL LETTER A (U+0041) followed by COMBINING RING ABOVE (U+030A).

While all of the above examples would be dealt with as stated above, regardless of the version of Unicode used by the server, the canonical equivalence relation is subject to change. This is because successive Unicode versions can add characters, creating instances of NFC form strings that did not exist previously.

In the context of NFSv4 servers, such equivalences can only be acted upon in the context of UTF8-aware file systems. In that context:

* Servers MAY map name strings to other canonically equivalent strings, so that the name of a file can be different from the name specified by the user.

Clients are expected to be tolerant of such mappings while many users are likely to consider canonically equivalent strings as being the same. Users who consider such strings as different would use UTF8-unaware file systems or those that did not modify user names.

* Servers MAY treat canonically equivalent strings as identical when searching for a given file without making any change in the names presented when the file is created.

  Clients are expected to be tolerant of such mappings while most users are likely to consider canonically equivalent strings as being the same. Users who consider these different would normally use UTF8-unaware file systems.

* While some other protocols deal with normalization issues by rejecting strings that are not in a particular normalization form, this option is not available to NFSv4 servers and NFsv4 clients are not required to abide by server-imposed normalization-form constraints

  Because the canonical equivalence relation can change, placing the burden of adapting to a particular normalization form and Unicode version would create a difficult-to-maintain file access API.

* Although clients can generally avoid any concern with the server's approach to normalization issues, there are, as described Section 7.3, some forms of client-side name caching for which the fact that the server treats two different strings as equivalent makes it desirable for the client do so as well, or not use those forms of name caching.

  Because of the current inability of the client to determine the Unicode version used by the server, such forms of name caching are best avoided when using UTF8-aware file systems However Appendix B.4 discusses available possibilities for providing restrictions on such forms of name caching without eliminating them.

  For a discussion of how the client might be made aware of the specific canonical equivalence relation used by the server, see Appendix A.4.

7.2. Handling of Case-insensitive Equivalence of Strings

In many environments it is desirable to treat two strings as equivalent if they differ only as to case. This need arises when using operating environments in which file names are treated in a case-insensitive manner. While determining whether two strings are equivalent except for case, can, in many environments, be a straightforward matter, there are, in internationalized environments, situations in which user language preference or other similar considerations require the server implementer to make choices in this regard. See Appendix A.1 for a discussion of these cases.

In the context of NFSv4 servers, such equivalences can only be acted upon in the context of UTF8-aware file systems. In that context:

* Servers MAY map a name string to another string equivalent except with regard to case, so that the name of a file can be different than the name requested by the user.

  When the OPTIONAL attributes case_insensitive and case_preserving are implemented, their values will both be false.

* Servers MAY treat name strings that only differ as to case as identical when searching for a given file without making any change in the name presented when the file is created.

  When the OPTIONAL attributes case_insensitive and case_preserving are implemented, their values will be true and false, respectively.

* Although clients can generally avoid any concern with the server's approach to case-handling issues, there are, as described Section 7.3, some forms of client-side name caching for which the fact that the server treats two different strings as equivalent make it desirable for the client do so as well.

  Because of the current inability of the client to find out the details of the case equivalence relation use by the server, such forms of name caching are best avoided when using case-insensitive file systems. However Appendix B.4 discusses available possibilities for providing restrictions on such forms of name caching without eliminating them.

  For a discussion of how the client might be made aware of the case-equivalence relation used by the server, see Appendix A.3.

7.3. String Equivalence and Client Name Caching

While most client functions are not affected by a server's implementation of various equivalence classes, there are a number of forms of name caching that require the client to be aware of string equivalence classes implemented by the server

* If the client implements negative name caching by caching the results of LOOKUP, OPEN, or ACCESS operations that find that the file does not exist, the server's treatment of two distinct strings as equivalent creates a potential problem.

  When negative name caching is implemented, there needs to be ways to eliminate records of the non-existence of particular files when they are no longer appropriate. This will occur when the files are found using LOOKUP, OPEN, or ACCESS or when names are added to the directory using OPEN, CREATE, LINK, or RENAME. When name equivalence relationships exist on the server, the client cannot act appropriately when files with previously non-existing names are found or created using distinct names considered equivalent.

* If the client uses the results of earlier READDIR operations to enable later LOOKUP operations to be avoided, the efficiency of that caching is undercut when the client is unaware of the details of these equivalence relations.

  In such situations, the client's cached READDIR entry cannot be used, as it would on the server, to satisfy a LOOKUP for a distinct name equivalent to the first, requiring an over-the-wire operation that such caching is intended to avoid.

Because of these issues, when name equivalences are in effect, the above forms of caching cannot work effectively and are best avoided.

8. Servers That Accept File Component Names That Are Not Valid UTF-8 Strings

Servers MAY accept, on all or on some subset of the underlying file systems exported, component names that are not valid UTF-8 strings.

A typical pattern is for a server to use UTF-8-unaware underlying file systems that treat component names as uninterpreted strings of bytes, rather than having any awareness of the character set being used.

Such servers MUST use an octet-by-octet comparison of component name strings to determine equivalence (as opposed to any broader notion of string comparison).

This is because the server has no knowledge of the specific character encoding being used.

9. The Attribute Fs_charset_cap

This OPTIONAL attribute, appears to have been added to NFSv4.1 to allow servers, while staying within the constraints of the stringprep-based specification of internationalization, to allow uses of UTF-8-unaware naming by clients. As a result, those NFSv4 servers implementing internationalization as NFSv3 had done, could be considered spec-compliant, as long as a later "SHOULD" was ignored. However, because use of UTF-8 was tied to existing stringprep restrictions, implementations of internationalization, that were aware of Unicode canonical equivalence issues were not provided for. Although this attribute may have been implemented despite the lack of need for two separate bits, the overall scheme was never implemented and NFSv4.1 implementations dealt with internationalization in the same way as NFSv4.0 implementations had.

The attribute still contains two flag bits although the motivation for having two bits remains unclear.

Section 9.1 replaces Section 14.4 of [RFC8881], taking into account the behavior of existing implementations of [RFC5661] [RFC8881] while providing best effort compatibility with the definition in [RFC5661] and [RFC8881].

9.1. The Attribute Fs_charset_cap Going Forward

```
const FSCHARSET_CAP4_CONTAINS_NON_UTF8  = 0x1;
const FSCHARSET_CAP4_ALLOWS_ONLY_UTF8   = 0x2;

typedef uint32_t        fs_charset_cap4;
```

This attribute provides a simple way of determining whether a particular file system behaves as a UTF-8-only server and rejects file names which are not valid UTF8-encoded strings. When this attribute is supported and the value returned has the FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 flag set, the error NFS4ERR_INVAL MUST be returned if any file name argument contains a string which is not a valid UTF8-encoded string.

When this attribute is supported and the value returned has the FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 flag clear, the error NFS4ERR_INVAL will not be returned based on the client’s adherence to the rules of UTF-8.

The FSCHARSET_CAP4_CONTAINS_NON_UTF8 flag exists for historical reasons only and has no clear behavior associated with it. Servers SHOULD set the value of this flag to the complement of the setting of the FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 flag.

Regarding the use of "SHOULD" above, the only valid reason to bypass the recommendation is the need to interact properly with an existing client that, based on previous unclear guidance, uses the FSCHARSET_CAP4_CONTAINS_NON_UTF8 flag to determine internationalization-related characteristics of the file system being accessed. When doing this, the server implementer needs to be aware that the previous lack of clear guidance may have caused other clients to behave incorrectly when the recommendation is bypassed.

Clients SHOULD ignore the FSCHARSET_CAP4_CONTAINS_NON_UTF8 flag.

Regarding the use of "SHOULD" above, the only valid reason to bypass the recommendation is the difficulty of changing, at this late date, previous implementation that interpreted previous specifications as mandating, in some way, that the server behavior type specified in Section 6, could be determined in this way.

When this attribute is not supported, the client can perform a LOOKUP using a name not conforming to the rules of UTF-8 and use the error returned to determine whether non-UTF-8 names are accepted.

10. String Encoding

Strings that potentially contain characters outside the ASCII range [RFC20] are generally represented in NFSv4 using the UTF-8 encoding [RFC3629] of Unicode [UNICODE]. See [RFC3629] for precise encoding and decoding rules.

Some details of the protocol treatment depend on the type of string:

* For strings that are component names, the preferred encoding for any non-ASCII characters, when the encoding is known by client and server, is the UTF-8 representation of Unicode.

  In many cases, clients have no knowledge of the encoding being used, with the encoding done at the user level under the control of a per-process locale specification. As a result, it is impossible in such cases for the NFSv4 client to enforce the use of UTF-8. The use of such encodings can be problematic, since it may interfere with access to files stored using other forms of name encoding. Also, normalization-related processing (see

Section 7.1) of a string not encoded in UTF-8 could result in inappropriate name modification or aliasing. In cases in which one has a non-UTF-8 encoded name that accidentally conforms to UTF-8 rules, substitution of canonically equivalent strings can change the non-UTF-8 encoded name drastically.

For similar reasons, where non-UTF-8 encoded names are accepted, case-related mappings cannot be relied upon. For this reason, the attribute case_insensitive MUST NOT be returned as TRUE for file systems which accept non-UTF-8 encoded file names.

The kinds of modification and aliasing mentioned here can lead to both false negatives and false positives, depending on the strings in question, which can result in security issues such as elevation of privilege and denial of service (see [RFC6943] for further discussion).

* For strings based on domain names, non-ASCII characters MUST be represented using the UTF-8 encoding of Unicode or some encoding based on that (e.g. xn-labels including Punycode), and additional string format restrictions will apply. See Section 11 for details.

* The contents of symbolic links (of type linktext4 in the XDR) MUST be treated as opaque data by NFSv4 servers. Although UTF-8 encoding is often used, it need not be. In this respect, the contents of symbolic links are like the contents of regular files in that their encoding is not within the scope of this specification.

* For other sorts of strings, any non-ASCII characters SHOULD be represented using the UTF-8 encoding of Unicode.

11. String Types with Processing Defined by Other Internet Areas

There are two types of strings that NFSv4 deals with that are based on domain names. Processing of such strings is defined by other standards-track documents, and hence the processing behavior for such strings should be consistent across all server and client operating systems and server file systems.

This section differs from other sections of this document in two respects:

* Although the normative statements within this section are derived from the behavior of existing NFSv4 implementations, they need to be consistent with existing RFCs regarding domain handling.

* Because of the switch from IDNA2003 [RFC3490] [RFC3491] to IDNA2008 [RFC5890], this section is necessarily different from the corresponding section (i.e. Section 12.6) of [RFC7530]. The differences are discussed in Section 11.1.

Because of this shift, there could be compatibility issues to be expected between implementations obeying Section 12.6 of [RFC7530], if any such implementations exist, and those following this document. Whether such compatibility issues actually exist depends on the behavior of NFSv4 implementations and how domain names are actually used in existing implementations. These matters will be discussed in Section 11.2.

The types of strings referred to above are as follows:

* Server names as they appear in the fs_locations and fs_locations_info attribute. Note that for most purposes, such server names will only be sent by the server to the client. The exception is the use of these attributes in a VERIFY or NVERIFY operation.

* Principal suffixes that are used to denote sets of users and groups, and are in the form of domain names. These may appear in the owner and group attributes and as who values within ACEs that appear within ACL-related attributes. Such values are sent by the client to the server in performing SETATTR, VERIFY, and NVERIFY operations and returned to the client in performing GETATTR operations.

There is likely to be few or no implementations conforming to Section 12.6) of [RFC7530] as a result of how internationalization was supported previously.

* When [RFC3530] was published, its discussion of internationalization was ignored as unimplementable and inappropriate. This included the handling of domain names, although the reasons for ignoring the specification might have been different in that case.

* When [RFC7530] was published, implementers saw no reason to modify the existing domain-handling code which worked adequately for valid domain names.

These strings can be expressed in two ways:

* As the UTF-8 representation of the string represented. This includes cases in which all of the characters are within the Ascii range. We refer to such representations as the U-label form.

* As the string "xn--" followed by the text of the string transformed using the Punycode encoding described in [RFC3492]. We refer to such representations as the xn-label form.

In cases in which such strings are sent by the client to the server:

* The server MUST accept such strings in xn-label form.

  When it does so, MAY reject, using the error NFS4ERR_INVAL, any of the following:

  - a string for which the characters after "xn--" are not valid output of the Punycode algorithm [RFC3492].

  - a string that contains a reserved LDH label (see [RFC5890]) which is not an XN-label.

* The server MAY accept such strings in U-label form and is REQUIRED to do so only in the case in which the string consists only of ascii characters.

  The server MAY reject, using the error NFS4ERR_INVAL, strings which are not valid UTF-8 or do not form a valid U-label for other reasons.

When the server does not make the validity checks mentioned above, the result will be use of an invalid domain name. Since such domains do not exist, clients are unlikely to use them and servers will be unable to access such domains.

Servers MUST NOT modify the string to a canonically equivalent one (e.g. as part of normalization-related processing). Further, changes of case SHOULD NOT be done at all and MUST NOT be done for strings that contain Unicode characters outside the ASCII range.

In cases in which such strings are sent by the server to the client, they MAY be presented in either form. In view of this, clients that anticipate receiving internationalized domain names will find it advisable to convert such strings to a common form, preferred by the client's users.

A domain name returned by GETATTR will generally be exactly the same as that presented by SETATTR. The following exceptions are possible:

* There is a change of case when the domain string does not contain any multi-byte Unicode characters.

* The server converts an xn-label string to the corresponding U-label string or vice versa.

For VERIFY and NVERIFY, additional string processing requirements apply to verification of the owner and owner_group attributes; see the section entitled "Interpreting owner and owner_group" for the document specifying the minor version in question (RFC7530 [RFC7530], RFC8881 [RFC8881])

11.1. Effect of IDNA Changes

Overall, the effect of the shift to IDNA2008 is to limit the degree of understanding of the IDNA-based restrictions on domain names that were expected of NFSv4 in RFC7530 [RFC7530]. Despite this specification, the degree to which implementations actually implemented such restrictions is open to question. The consequences of this uncertainty will be discussed in detail in Section 11.2.

In analyzing how various cases are to be dealt with according to RFC7530, there a number of troubling uncertainties that arise in trying to interpret the existing specification:

* There are a number of cases in which "SHOULD" is used that are confusing. According to RFC2119 [RFC2119], "SHOULD" means that "there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course". To fully understand a particular "SHOULD", there needs to be enough context to determine whether particular reasons for ignoring the item are in fact valid, and sufficient guidance to understand the implication of ignoring the item. In the absence of such information, the relevant fact is that the peer needs to deal with the item being ignored, making the implications of a "SHOULD" hard to distinguish from those of "MAY".

* While the document states, "the general rules for handling all of these domain-related strings are similar and independent of the role of the sender or receiver as client or server", all of the following text is explicitly about the server’s options, choices and responsibilities, leaving the client case unclear.

* In a number of places within the paragraph describing server approach #1, the word "can" is used as in the text "the server can use the ToUnicode function", leaving it unclear whether the server can choose to do anything else and if so what.

The following cases are those where RFC7530 requires use of IDNA handling and this requirement could, if implementations follow them, create potential compatibility issues, which need to be understood.

* The degree to which RFC3490 [RFC3490] requires that characters other than U+002E (full stop) be treated as label separators, including U+3002 (ideographic full stop), U+FF0E (fullwidth full stop), U+FF61 (halfwidth ideographic full stop).

* The degree to which RFC3490 [RFC3490] might require that server or client needs to validate a putative A-label or U-label or to rectify it if it is not valid.

11.2. Potential Compatibility Issues Related to IDNA Changes

There are a number of factors relating to the handling of domain names within NFSv4 implementations that are important in understanding why any compatibility issues might be less troubling than a comparison of the two IDNA approaches might suggest:

* Much of the potentially conflicting IDNA-related behavior required or recommended for the server by RFC7530 [RFC7530] appears to not be actually implemented, limiting the potential harmful effects of ceasing to mandate it.

* Even if such behavior were implemented by servers, no compatibility issue would arise unless clients actually relied on the server to implement it. Given that none of this behavior is made required, the chances of that occurring is quite small.

* The range of potential values for user and group attributes sent by clients are often quite small with implementations commonly restricting all such values to a single domain string. This is even though RFCs 7530 [RFC7530] and 8811 [RFC8881] are written without mention of such restrictions.

  Specification of users and groups in the "id@domain" format within NFSv4 was adopted to enable expansion of the spaces of users and groups beyond the 32-bit id spaces mandated in NFSv3 [RFC1813] and NFsv2 [RFC1094]. While one obstacle to expansion was eliminated, most implementations were unable to actually effect that expansion, principally because the underlying file systems used assume that user and group identifiers fit in 32 bits each and the vnode interfaces used by server implementations make similar assumptions.

Given these restrictions, the typical implementation pattern is for servers to accept only a single domain, specified as part of the server configuration, together with information necessary to effect the appropriate name-to-id mappings.

* For the other uses of domain names in NFSv4, to represent host names in location attributes, the values are generated by the server and will normally only include host names within DNS-registered domains.

Keeping the above in mind, we can see that interoperability issues, while they might exist, are unlikely to raise major challenges as looking to the following specific cases shows.

* When an internationalized domain name is used as part of a user or group, it would need to be configured as such, with the domain string known to both client and server.

  While it is theoretically possible that a client might work with an invalid domain string and rely on the server to correct it to an IDNA-acceptable one, such a scenario has to be considered extremely unlikely, since it would depend on multiple servers implementing the same correction, especially since there is no evidence of such corrections ever having been implemented by NFSv4 servers.

* When an internationalized domain in a location string is meant to specify a registered domain, similar considerations apply.

  While it is theoretically possible that a client might work with an invalid domain string and rely on the server to correct it to an appropriate registered one, such a scenario has to be considered extremely unlikely, since it would depend on multiple servers implementing the same correction, especially since there is no evidence of such corrections ever having been implemented by NFSv4 servers.

* When an internationalized domain in a location string is meant to specify a non-registered domain, any such server-applied corrections would be useless.

  In this situation, any potential interoperability issue would arise from rejecting the name, which has to be considered as what should have been done in the first place.

12. Errors Related to UTF-8

Where the client sends an invalid UTF-8 string, the server MAY return an NFS4ERR_INVAL error. This includes cases in which inappropriate prefixes are detected and where the count includes trailing bytes that do not constitute a full Multiple-Octet Coded Universal Character Set (UCS) character.

Requirements for server handling of component names that are not valid UTF-8, when a server does not return NFS4ERR_INVAL in response to receiving them, are described in Section 8.

Where the string supplied by the client is not rejected with NFS4ERR_INVAL but contains characters that are not supported by that server as a value for that string (e.g., names containing slashes, characters that the particular file system are not appropriate in names, or characters that do not fit into 16 bits when converted from UTF-8 to a Unicode codepoint), the server MUST indicate such a rejection using an NFS4ERR_BADCHAR error.

Where a UTF-8 string is used as a file name, and the file system, while supporting all of the characters within the name, does not allow that particular name to be used, the server will return the error NFS4ERR_BADNAME. This includes such situations as file system prohibitions of "." and ".." as file names for certain operations, and similar constraints.

In making such the determinations discussed above, servers are depending on the character encoding used even when the encoding using UTF-8 is not enforced. Since such rejections are limited to characters whose values are below 128, clients are, as a practical matter, safe if their encodings are consistent with UTF-8 in the handling of byte values 127 and below.

13. IANA Considerations

The current document does not require any actions by IANA.

14. Security Considerations

Unicode in the form of UTF-8 is generally used for file component names (i.e., both directory and file components). However, other character sets may also be allowed for these names. For the owner and owner_group attributes and other sorts strings whose form is affected by standards outside NFSv4 (see Section 11.) are always encoded as UTF-8. String processing (e.g., Unicode normalization) raises security concerns for string comparison. See Sections 11 and 7 as well as the respective Sections 5.9 of RFC7530 [RFC7530] and

RFC8881 [RFC8881] for further discussion. See [RFC6943] for related identifier comparison security considerations. File component names are identifiers with respect to the identifier comparison discussion in [RFC6943] because they are sed to identify the objects to which ACLs are applied (See the respective Sections 6 of RFC7530 [RFC7530] and RFC8881 [RFC8881]).

Note that the references to per-minor-version documents may become out-of-date as part of the rfc5661bis effort. In the event that happens, it will be necessary for users to consult RFCs derived from [I-D.dnoveck-nfsv4-security] and [I-D.dnoveck-nfsv4-acls].

15. References

15.1. Normative References

[RFC20] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, October 1969, <http://www.rfc-editor.org/info/rfc20>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, DOI 10.17487/RFC3492, March 2003, <https://www.rfc-editor.org/info/rfc3492>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <https://www.rfc-editor.org/info/rfc7530>.

[RFC7862] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 Protocol", RFC 7862, DOI 10.17487/RFC7862, November 2016, <https://www.rfc-editor.org/info/rfc7862>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8178] Noveck, D., "Rules for NFSv4 Extensions and Minor Versions", RFC 8178, DOI 10.17487/RFC8178, July 2017, <https://www.rfc-editor.org/info/rfc8178>.

[RFC8881] Noveck, D., Ed. and C. Lever, "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 8881, DOI 10.17487/RFC8881, August 2020, <https://www.rfc-editor.org/info/rfc8881>.

[UNICODE] The Unicode Consortium, "The Unicode Standard, Version 7.0.0", (Mountain View, CA: The Unicode Consortium, 2014 ISBN 978-1-936213-09-2), June 2014, <http://www.unicode.org/versions/Unicode7.0.0/>.

[UNICODE-CASEF] The Unicode Consortium, "CaseFolding-13.0.0.txt", (Mountain View, CA: The Unicode Consortium, 2014 ISBN 978-1-936213-26-9), March 2020, <https://www.unicode.org/Public/13.0.0/ucd/CaseFolding.txt>.

[UNICODE-CASEM] The Unicode Consortium, "The Unicode Standard, Version 13.0.0, Section 5.18 Case Mappings", (Mountain View, CA: The Unicode Consortium, 2014 ISBN 978-1-936213-26-9), March 2020, <http://www.unicode.org/versions/Unicode13.0.0/ch05.pdf#G21180>.

15.2. Informative References

[I-D.dnoveck-nfsv4-acls] Noveck, D., "ACLs within the NFSv4 Protocols", Work in Progress, Internet-Draft, draft-dnoveck-nfsv4-acls-06, 23 February 2025, <https://datatracker.ietf.org/doc/html/draft-dnoveck-nfsv4-acls-06>.

[I-D.dnoveck-nfsv4-security] Noveck, D., "Security for the NFSv4 Protocols", Work in Progress, Internet-Draft, draft-dnoveck-nfsv4-security-11, 29 August 2024, <https://datatracker.ietf.org/doc/html/draft-dnoveck-nfsv4-security-11>.

[I-D.ietf-nfsv4-rfc3010bis]
Beame, C., Thurlow, R., Callaghan, B., Robinson, D., Noveck, D., Eisler, M., and S. Shepler, "Network File System (NFS) version 4 Protocol", Work in Progress, Internet-Draft, draft-ietf-nfsv4-rfc3010bis-05, 7 November 2002, <https://datatracker.ietf.org/doc/html/draft-ietf-nfsv4-rfc3010bis-05>.

[RFC1094] Nowicki, B., "NFS: Network File System Protocol specification", RFC 1094, DOI 10.17487/RFC1094, March 1989, <https://www.rfc-editor.org/info/rfc1094>.

[RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, DOI 10.17487/RFC1813, June 1995, <https://www.rfc-editor.org/info/rfc1813>.

[RFC3010] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "NFS version 4 Protocol", RFC 3010, DOI 10.17487/RFC3010, December 2000, <https://www.rfc-editor.org/info/rfc3010>.

[RFC3454] Hoffman, P. and M. Blanchet, "Preparation of Internationalized Strings ("stringprep")", RFC 3454, DOI 10.17487/RFC3454, December 2002, <https://www.rfc-editor.org/info/rfc3454>.

[RFC3490] Faltstrom, P., Hoffman, P., and A. Costello, "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, DOI 10.17487/RFC3490, March 2003, <https://www.rfc-editor.org/info/rfc3490>.

[RFC3491] Hoffman, P. and M. Blanchet, "Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)", RFC 3491, DOI 10.17487/RFC3491, March 2003, <https://www.rfc-editor.org/info/rfc3491>.

[RFC3530] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", RFC 3530, DOI 10.17487/RFC3530, April 2003, <https://www.rfc-editor.org/info/rfc3530>.

[RFC5661] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 5661, DOI 10.17487/RFC5661, January 2010, <https://www.rfc-editor.org/info/rfc5661>.

[RFC6365] Hoffman, P. and J. Klensin, "Terminology Used in Internationalization in the IETF", BCP 166, RFC 6365, DOI 10.17487/RFC6365, September 2011, <https://www.rfc-editor.org/info/rfc6365>.

[RFC6943] Thaler, D., Ed., "Issues in Identifier Comparison for Security Purposes", RFC 6943, DOI 10.17487/RFC6943, May 2013, <https://www.rfc-editor.org/info/rfc6943>.

Appendix A. Providing Information about Server Choices Regarding String Equivalence

A.1. Important Issues for Case-insensitive Handling of File Names

In this section, we discuss many of the interesting and/or troublesome issues that the need for case-insensitive handling gives rise to in fully internationalized environments. Many of these are also discussed in [UNICODE-CASEM]. However, our treatment of these issues, while not inconsistent with that in [UNICODE-CASEM], differs significantly for a number of reasons:

* Our primary focus is on case-insensitive string comparison rather than with case mapping per se. While such comparison is natural for the client and allowed for servers, its greater flexibility makes it important to understand its capabilities in dealing with potentially troublesome issues in providing case-insensitive file name handling.

* Because a case mapping model forces the specification of a single case mapping result when there are multiple potentially valid results, there are inevitably cases in which the result chosen is inappropriate for some users. These are cases in which F-type and S-type mappings are present and in which C-type and T-type mappings conflict. Normally, an appropriate choice is selected by use of the locale, but in a file system environment, valid locale information might not be present. As a result, case-insensitive string comparison, which does not force such case mapping choices, will be more desirable since it allows construction of sets of equivalent strings based on multiple mappings which is not possible when case mapping is the goal.

The examples below present common situations that go beyond the simple invertible case mappings of Latin characters and the straightforward adaptation of that model to Greek and Cyrillic. In EX4 and EX5 we have case-based sets of equivalent strings including multi-character strings not derived from canonical equivalences while for EX7 and EX8 all multi-character strings are derived from canonical equivalences. In addition, EX1, EX2, EX3 and EX6 discuss other situations in which a set of equivalent strings has more than two elements.

EX1: Certain digraph characters such LATIN SMALL LETTER DZ (U+01F3) have additional case variants to consider such as the title case character LATIN CAPTAL LETTER D WITH SMALL LETTER Z (U+01F2) in addition to the uppercase LATIN CAPITAL LETTER DZ (U+01F1). While the variant for title case would not appear in names in case-insensitive non-case-preserving file systems, case-insensitive string comparison has no problem in treating these three characters as within same se of equivalent characters.

This set of equivalent strings can be derived using only C-type mappings. The possibility of mapping these characters to the two-character sequences they represent is not a troublesome issue since that would be derived from a compatibility equivalence, rather than a canonical equivalence, and there is no F-type mapping making it an option.

EX2: To deal with the case of the OHM SIGN (U+2126) which is essentially identical to the GREEK CAPITAL LETTER OMEGA (U+03A9), one can construct an set of equivalent characters consisting of OHM SIGN (U+2126), GREEK CAPITAL LETTER OMEGA (U+03A9), and GREEK SMALL LETTER OMEGA (U+03C9).

This set of equivalent strings can be derived using only C-type mappings. Both OHM SIGN (U+2126), and GREEK CAPITAL LETTER OMEGA (U+03A9) lowercase to GREEK LETTER OMEGA (U+03C9), while that character only uppercases to GREEK CAPITAL LETTER OMEGA (U+03A9).

EX3: To deal with the case of the ANGSTROM SIGN (U+212B) which is essentially identical to LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5), one can construct a set of equivalent strings consisting of ANGSTROM SIGN (U+212B), LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5), LATIN SMALL LETTER A WITH RING ABOVE (U+00E5), together with the two-character sequences involving LATIN CAPITAL LETTER A (U+0041) or LATIN SMALL LETTER A (U+0061) followed by COMBINING RING ABOVE (U+030A).

This set of equivalent strings can be derived using only C-type mappings together with the ability to map characters to canonically equivalent strings. Both ANGSTROM SIGN (U+212B), and LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5) lowercase to LATIN SMALL LETTER A WITH RING ABOVE (U+00E5), while that character only uppercases to CAPITAL LETTER A WITH RING ABOVE (U+00C5).

EX4: In some cases, case mapping of a single character will result in a multi-character string. For example, the German character LATIN SMALL LETTER SHARP S (U+00DF) would be uppercased to "SS", i.e. two copies of LATIN CAPITAL LETTER S (U+0053). On the other hand, in some situations, it would be uppercased to the character LATIN CAPITAL LETTER SHARP S (U+1E9E), using an S-type mapping, referred to as an instance of "Tailored Casing". Unfortunately, in the context of a file system, there is unlikely to be available information that provides guidance about which of these case mappings should be chosen. However, the use of case-insensitive mappings with larger equivalence classes often provides handling that is acceptable to a wider variety of users. In this case, if both mappings were used together to create a set of equivalent strings, German-speakers would get the mapping they expect while those unfamiliar with these characters only see them when they access a file whose name contains such characters.

It appears that if the construction of case-based equivalence classes were generalized to include multi-character sequences, then all of LATIN SMALL LETTER SHARP S (U+00DF), LATIN CAPITAL LETTER SHARP S (U+1E9E), "ss", "sS", "Ss", and "SS" would belong to the same equivalence class and could be handled by the general algorithm described in Appendix B.1, rather than by code specifically written to deal with this particular issue, which might hard to maintain.

EX5: Other ligatures, such as LATIN SMALL LIGATURE FFL (U+FB04), could be handled similarly by this algorithm, if there were felt to be a need to do so. However, because the decomposition of this character into the string consisting of the three letters LATIN SMALL LETTER F (U+0066), LATIN SMALL LETTER F (U+0066), LATIN SMALL LETTER L (U+006C), is a compatibility equivalence, and the F-type mapping of this ligature to the three constituent characters is to be treated as optional, implementations can choose either to treat this character as having no uppercase equivalent or treat it as part of larger set of equivalent strings including "ffl", "ffL", "fFl", etc.).

EX6: The character COMBINING GREEK YPOGEGRAMMENI (U+0345), also known as "iota-subscript" requires special handling when uppercasing and lowercasing. While the description of the appropriate handling for this character, in the case mapping section, is focused on multi- character sequences representing diphthongs, case-insensitive comparisons can be performed without consideration of multi-character sequences. This can be done by assigning COMBINING GREEK YPOGEGRAMMENI (U+0345), GREEK SMALL LETTER IOTA (U+03B9), and GREEK CAPITAL LETTER IOTA (U+0399) to the same equivalence class, even though the first of these is a combining character and the others are not.

EX7: In some cases, context-dependent case mapping is required. For example, GREEK CAPITAL LETTER SIGMA (U+03A3) lowercases to GREEK SMALL LETTER SIGMA (U+03C3) if it is followed by another letter and to GREEK SMALL LETTER FINAL SIGMA (U+03C2) if it is not.

Despite this, case-insensitive comparisons can be implemented, by considering all of these characters as part of the same equivalence class, without any context-dependence, and this set of equivalent strings can be derived using only C-type mappings.

EX8: In most languages written using Latin characters, the uppercase and lowercase varieties of the letter "I" map to one another. In a number of Turkic languages, there are two distinct characters derived from "I" which differ only with regard to the presence or absence of a dot so that there are both capital and small i’s with each having dotted and dotless variants. Within such languages, the dotted and dotless I’s represent different vowel sounds and are treated as separate characters with respect to case mapping. The uppercase of LATIN SMALL LETTER I (U+0069) is LATIN CAPITAL LETTER I WITH DOT ABOVE (U+0130), rather than LATIN CAPITAL LETTER I (U+0049). Similarly the lowercase of LATIN CAPITAL LETTER I (U+0049) is LATIN SMALL LETTER DOTLESS I (U+0131) rather than LATIN SMALL LETTER I (U+0069).

When doing case mapping, the server must choose to uppercase LATIN SMALL LETTER I (U+0069) to either LATIN CAPITAL LETTER I (U+0049), based on a C-type mapping to LATIN CAPITAL LETTER I WITH DOT ABOVE (U+0130), based on a T-type mapping. The former is acceptable to most people but confusing to speakers of the Turkic languages in question since the case mapping changes the character to represent a different vowel sound. On the other hand, the latter mapping seemingly inexplicably results in a character many users have never seen before. Normally such choices are dealt with based on a locale but, in a file system environment, no locale information is likely to be available.

In the context of case-insensitive string comparison, it is possible to create a larger set of equivalent strings, including all of the letters LATIN SMALL LETTER I (U+0069), LATIN CAPITAL LETTER I (U+0049), LATIN CAPITAL LETTER I WITH DOT ABOVE (U+0130), LATIN SMALL LETTER DOTLESS I (U+0131) together with the two-character string consisting of LATIN CAPITAL LETTER I (U+0049) followed by COMBINING DOT ABOVE (U+0307).

A.2. Defining Case-Insensitive Processing of File Names

When a server implements case-insensitive file name handling, it is desirable that clients do so as well. For example, if a client possessing the cached contents of a directory, notes that the file "a" does not exist, it cannot immediately act on that presumed non-existence, without checking for the potential existence of "A" as well. As a result, clients, in order to do certain form of name caching, might need to be able to provide case-insensitive name comparisons, irrespective of whether the server handling is case-preserving or not.

Because case-insensitive name comparisons are not always as straightforward as the above example suggests, the client, if it is to emulate the server’s name handling, would need information about how certain cases are to be dealt with. In cases in which that information is unavailable, the client needs to avoid making assumptions about the server’s handling, since it will be unaware of the Unicode version implemented by the server, or many of the details of specific issues that might need to be addressed differently by different server file systems in implementing case-insensitive name handling.

Many of the problematic issues with regard to the case-insensitive handling of names are discussed in Section 5.18 of the Unicode Standard [UNICODE-CASEM] which deals with case mapping. While we need to address all of these issues as well, our approach will not be exactly the same.

* Since the client would only need to be doing case-insensitive comparisons, issues that apply only to uppercasing or lowercasing do not have the same significance.

* Many clients will have to operate correctly even in the absence of detailed information about the specifics of server-side case-mapping or the version of Unicode implemented by the server.

* Clients will have to accommodate server behaviors not anticipated by the Unicode Specification since it might be that neither the server nor the client would have any relevant locale knowledge when file names are processed.

Another source of information about case-folding, and indirectly about case-insensitive comparisons, is the case-folding text file which is part of the Unicode Standard [UNICODE-CASEF]. This file contains, for each Unicode character that can be uppercased or lowercased, a single character, or, in some cases a string of characters of the other case. For characters in capital case, the lowercase counterpart is given. Each of the mappings is characterized as of one of four types:

* Common case folding, denoted by a status field of "C". These are used for mapping where a single character can be mapped to a single character of another case. These are always valid with one potential exception being the mappings of LATIN CAPITAL LETTER I to LATIN SMALL LETTER I and vice versa, which might be superseded by the T-type mappings associated with some Turkic languages when written using Latin letters.

* Full case folding, denoted by a status field of "F". These are used for mappings in which single character is mapped to a multi-character string of a different case.

* Special case folding, denoted by a status field of "S". These provide additional single-character-to-single-character which might be used when there is also an F-type mapping of the same character. In the case of case folding, this is an alternative to the corresponding F-type, although, for the purposes of case-insensitive string comparison, it is possible for both to be considered valid at the same time

* Special case foldings for Turkic languages, denoted by a status field of "T". These consist of the invertible case mappings between LATIN SMALL LETTER I (U+0069) and LATIN CAPITAL LETTER I WITH DOT ABOVE (U+0130) and between LATIN CAPITAL LETTER I (U+0049) and LATIN SMALL LETTER DOTLESS I (U+0131). The relationship between these mappings and the C-type mappings for LETTER I is discussed below in item EX8.

While the case mapping section does discuss case-insensitive string comparisons, and describes a procedure for constructing equivalence classes of Unicode characters, the description does not deal clearly with the effect of F-type mappings. There are a number of problems with dealing with F-type mappings for case folding and basing case-insensitive string comparisons on those mappings, particularly in situations, such as file systems, in which extensive processing of strings is unlikely to be practical.

* Mappings from single characters to multi-character strings, are, for case-folding purposes, not invertible. However, case-insensitive name comparison, by its nature, requires invertible mappings, in which a multi-character string is mapped to a single character of a different case. This is not compatible with any existing simple case-mapping model.

* Scanning of names for multi-character sequences might well be too complicated for effective implementation within a file system, especially since such sequences might overlap in complicated ways.

* Case foldings which map single characters to multi-character sequences (see item EX4 below for an important example), would give rise to very large sets of strings. This is because of the invertibility of case mappings when used to determine case-insensitive string equivalence. For example, a string of eight copies of the letter S would give rise to a set of 256 equivalent strings plus over two thousand others when the German SHARP S characters discussed in item EX4 are included.

Despite these potential difficulties, case mappings involving multi-character sequences can be reversed when used as a basis for case-insensitive string comparisons and incorporated into a set of equivalence classes on name strings, as described below.

* Case-insensitive servers MAY do either case-mapping to a chosen case (the non-case-preserving case), or case-insensitive string comparisons when providing a case-preserving implementation. In either case, the server MAY include F-type mappings, which map a single character to a multi-character string. However, only the case in which it is doing case-insensitive string comparison will it use the inverse of F-type mappings, in which a multi-character string is mapped to a single character of a different case

  In these cases, the server can choose to use either a C-type mapping or an F-type mapping, or both, when both exist. Similarly the server may choose to implement the C-type mappings of LATIN CAPITAL LETTER I to LATIN SMALL LETTER I and vice versa, the corresponding T-type mappings or both, although using only the T-type mappings is undesirable, unless there is a means of informing the client that it has been chosen, since users might reasonably expect LATIN CAPITAL LETTER I and LATIN SMALL LETTER I to treated identically in a case-insensitive file system.

* The client, when informed of the details of the client's handling of case, has the ability to efficiently implement an appropriate case-insensitive name comparison compatible with that of the server. This includes the ability to handle mappings between single characters and multi-character strings.

* Implementation of case-insensitive name comparisons will typically require a case-insensitive name hash.

A.3. Providing Information about Server Case-Insensitive Comparisons

It is possible to provide, as part of a valid NFSv4 extension, information sufficient to allow the client to be aware of, and potentially to emulate, case-insensitive comparisons implemented by the server. Such information would take the form of an OPTIONAL read-only per-fs file attribute. The information listed below would need to be included.

Whenever the value provided for a particular file system is invalid in some way, the client is justified in ignoring the attribute and acting as if it were not supported on that file system

* An integer denoting the version of Unicode on which the implemented case-equivalence relation was based.

The value zero would be available for use to indicate that the version is not relevant, either because the file system in question is UTF8-unaware, or because there is no server processing based on this version when the server is not case-insensitive and does not provide any normalization-related services.

If the value zero is received on a case-insensitive file system, the attribute value is considered invalid.

* Information regarding the special mapping for languages in which dot and dotless i's represent different vowel sounds (e.g. Turkish and Azeri).

  This could take the form of an enumeration having the values listed below, with any other value causing the attribute to be considered invalid.

  - A value indicating that only the C-type mapping are to be used in handling all i characters.

    In the case, LATIN SMALL LETTER I (U+0069) and LATIN CAPITAL LETTER I (U+0049) are considered case-equivalent while neither LATIN CAPITAL LETTER I WITH DOT ABOVE (U+0130) nor LATIN SMALL LETTER DOTLESS I (U+0131) are considered case-equivalent to any other character.

  - A value indicating that only the T-type mappings are to be used in handling all i characters.

    In this case, LATIN SMALL LETTER DOTLESS I (U+0131) is considered case-equivalent to LATIN CAPITAL LETTER I (U+0049) while neither LATIN CAPITAL LETTER I (U+0049) nor LATIN CAPITAL LETTER I WITH DOT ABOVE (U+0130) are considered case-equivalent to any other character.

  - A value indicating that both C-type and T-type mappings are to be used when handling i character.

    This value must not be used for file system that are case-insensitive but not case-preserving.

    In this case, all of LATIN SMALL LETTER I (U+0069), LATIN CAPITAL LETTER I (U+0049), LATIN SMALL LETTER DOTLESS I (U+0131), and LATIN CAPITAL LETTER I WITH DOT ABOVE (U+0130) are considered case-equivalent.

* Handling for special and full case foldings, as described in Appendix A.2.

This might take the form of a variable-length array of item of charfoldtype4, one for each character that can be subject to either S-type or F-type mappings. A possible realization of this type is described below. If this array is not of length zero and the Unicode version is zero, the attribute is considered invalid.

Each charfoldtype4 would contain the following:

* The numeric value of the UCS character, as opposed to the UTF-8 encoding of that character.

  If the character is one that has neither an S-type nor an F-type mapping, the attribute is considered invalid.

* A word with two bits, each of which indicates whether one of the two types of mapping are to be used in constructing sets of equivalent strings, with the low-order bit referring to S-type mappings and the next bit referring to F-type mappings. Depending on these bit settings, these mappings are either included or not in the set of case-equivalent strings associated with the particular character on the current the file system. This is in addition to any equivalences resulting from C-type mappings

  When either of these bits is set and the specified mapping does not exist for the associated character, the attribute is considered invalid.

If there are characters within the specified Unicode version that have S-type or F-type mappings specified and are not included in the array, then the equivalence set memberships for that character depend only on C-type mappings, if present.

A.4. Providing Information about Server Form-Insensitive Comparisons

It is possible to provide, as part of a valid NFSv4 extension, information sufficient to allow the client to be aware of, and potentially to emulate, form-insensitive comparisons implemented by the server. Such information would take the form of an OPTIONAL read-only per-fs file attribute. The following information would need to be included.

* An integer denoting the version of Unicode on which the implemented canonical equivalence was based.

  The value zero would be available for use to indicate that the version is not relevant, either because the file system in question is UTF8-unaware, or because there is no server processing based on the canonical equivalence relation.

* An enumerated value indicates whether names are mapped to their NFC or NFD equivalents, or compared in a form-insensitive manner without modification.

Although the attribute discussed in Appendix A.3 contains the Unicode version, allowing this one to be dispensed with, it is defined separately for the following reasons:

* Because of the additional effort in defining an attribute capable of supporting case-insensitivity and the low level of interest in that feature, the Working Group might decide to define this one first.

* Even when they were both defined some servers might choose not to support the one only applicable to a case-insensitive environment.

Appendix B. Implementation Discussions

B.1. Implementing Case-Insensitive Comparison of File Names

Implementing case-insensitive string comparisons based on equivalence classes including multi-character strings can be performed as described below. When such case-based set of equivalent strings contain multi-character strings, there are potential complexities that derive from the need to recognize such multi-character strings within the strings being compared.

The algorithm presented in this section requires the following for each set of equivalent strings:

(1): That if there is more than one multi-character string within the set of equivalent strings, the equivalence of those strings must be derivable from case-insensitive string equivalence using sets of equivalent strings each of whose members consist only of single-character strings.

(2): That each such set contains at least one single-character string.

Although other sources are possible (see items EX2 and EX3 in Appendix A.1), an important reason that multi-character sequences appear in case-insensitive sets of equivalent strings result from canonical decomposition of one or more precomposed characters. In such cases, elements of a case-insensitive equivalence class will include multiple characters because of the canonical decomposition of a single character.

While the algorithm presented in this section can deal with certain case-based equivalences deriving from canonical decomposition, it is not capable of providing general handling of the combination of canonical equivalence and case-based equivalence. While this can be addressed by normalizing strings before doing case-insensitive comparison, it is more efficient to do a general form-insensitive and case-insensitive string comparison in a single step as described in Appendix B.2

The following tables would be used by the comparison algorithm presented below.

* For each possible character value, the associated set of equivalent strings for case-insensitive comparison would be identified

* For each such set, the hash value contribution will be provided. In the case of set of equivalent strings that do not include multi-character strings including set that only include a single (single-character) member, this will be the hash value contribution of one particular variant (usually lower case) of the character

* In the case of set of equivalent string that do include multi-character strings, the hash value contribution needs to be equivalent to the combined contribution of each character within the multi-character string. In addition, for each such equivalence class, the length of the multicharacter string will be provided together with a pointer to an array describing the multi-character string, most probably presenting each character by a value of a case-equivalent character, most probably the lower-case variant.

Case-insensitive comparison proceeds as follows:

* Implementation of case-insensitive name comparisons will typically require a case-insensitive name hash using the tables described above. If such a hash value is kept for all cached names, comparisons of hashes can be used instead of the detailed comparison set forth below. Using such hash comparisons, a large set of potentially equivalent names can be excluded based on the occurrence of hash mismatches, since case-equivalent names would have the same hash value. value.

* For names with matching hash values, a detailed case-insensitive comparison will be necessary. This can proceed character-by-character or byte-by-byte. However, in the byte-by-byte case, processing in the event of a mismatch must start at the start of the current character, rather than the byte at which the difference was detected.

* In cases in which there is a mismatch, the associated equivalence classes will be compared. When these are identical, indicating the case equivalence of the two characters, the comparison of the two strings continues at the next character of each string.

* When the two equivalence classes are not identical, further comparisons to determine if a single character within one string matches (except for case) a multi-character string within the other. For each of two equivalence classes being compared that include a multi-character string, the check below must be made to determine whether the multi-character string at the corresponding position of the other string being compared, is within the current equivalence class. If neither of the two equivalence classes include multi-character strings, the comparison terminates with a mismatch indication.

* For each equivalence class that does include a multi-character string (there might be one or two), a scan needs to be made to see of the characters at the current position if the other string matches (except for case) the multi-character string which is included in the current equivalence class. If this check succeeds, for either equivalence class, the comparison of the two strings continues at the next character of each string. In the event of failure, the same sort of comparison is done using the other current equivalence class, if it include multi-character strings. Once this check fails for all equivalence classes that include multi-character strings, the comparison terminates with a mismatch indication.

B.2. Form-insensitive String Comparisons

This section deals with two varieties of form-insensitive string comparison:

* Providing a comparison function which is form-insensitive only. For any string, whether normalized or not, this function will determine it to be equivalent to all canonically equivalent strings, including but not limited, to the normalized forms NFC and NFD

* Providing a comparison function which is both form-insensitive and case-insensitive. This function will determine strings that only differ in case to be equal but will also be form-insensitive, as described above.

The non-normative guidance provided in this Appendix is intended to be helpful in dealing with two distinct implementation areas:

* Implementation of server-side file systems intended to be accessed as UTF8-aware file systems using NFSv4 protocols. While it is often the case that such file systems are developed by separate organizations from those concerned with NFSv4 server development, the internationalization- related requirements specified in this document must be adhered to for successful inter-operation when using UTF8-aware file systems, making this implementation guidance apropos despite any potential organizational barriers.

* Implementation of NFSv4 clients that might need to provide matching internationalization-related handling for reason discussed in Section 7.3.

There are three basic reasons that two strings being compared might be canonically equivalent even though not identical. For each such reason, the implementation will be similar in the cases in which form-insensitive comparison (only) is being done and in which the comparison is both case-insensitive and form- insensitive.

* Two strings may differ only because each has a different one of two code points that are essentially the same. Three code points assigned to represent units, are essentially equivalent to the character denoting those units. For example, the OHM SIGN (U+2126) is essentially identical to the GREEK CAPITAL LETTER OMEGA (U+03A9) as MICRO SIGN (U+00B5) is to GREEK SMALL LETTER MU (U+03BC) and ANGSTROM SIGN (U+212B) is to LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5).

  As discussed in items EX2 and EX3 in Appendix A.1, it is possible to adjust for this situation using tables designed to resolve case-insensitive equivalence, essentially treating the unit symbols as an additional case variant, essentially ignoring the fact that the graphic representation is the same. As a result, those doing string comparisons that are both form-insensitive and case-insensitive do not need to address this issue as part of form-insensitivity, since it would be dealt with by existing case-insensitive comparison logic.

Where there is no case-insensitive comparison logic, this function needs to be performed using similar tables whose primary function is to provide the decomposition of precomposed characters, as described in Appendix B.2.2.

* Two strings may differ in that one has the decomposed form consisting of a base character and an associated combining character while the other has a precomposed character equivalent.

  Although, as discussed in items EX3 in Appendix A.1, it is possible to use tables designed to resolve case-insensitive equivalence by providing as possible case-insensitively equivalent string, multi-character string providing the decomposition of precomposed characters, special logic to do so is only necessary when the decomposition is not a canonical one, i.e. it is a compatibility equivalence.

  In general, the table used to do comparisons, whether case-sensitive or not, needs to provide information about the canonical decomposition of precomposed characters. See Appendix B.2.2 for details.

* Two strings may differ in that the strings consist of combining characters that have the same effect differ as to the order in which the characters appear. For example, a letter might be followed by a combining character above and a combining character below and the combining characters might appear in different orders.

  There is no way this function could be performed within code primarily devoted to case-insensitive equivalence. However, this function could be added to implementations, providing both sorts of equivalence once it is determined that the base characters are case-equivalent while there is a difference of combining characters in to be resolved. (See Appendix B.2.5 for a discussion of how sets of combining characters can be compared).

B.2.1. Name Hashes

We discussed in Appendix B.1 the construction of a case-insensitive file name hash. While such a hash could also be form-insensitive if the hash contribution of every pre-composed character matched the combined contribution of the characters that it decomposes into.

However, there is no obvious way that sort of hash could respect the canonical equivalence of multiple combining characters modifying the same base character, when those combining characters appear in different orders. Addressing that issue would require a

significantly different sort of hash, in which combining characters are treated differently from others, so that the re-ordering of a string of combining characters applying to the same base character will not affect the hash.

In the hash discussed in Appendix B.1, there is no guarantee that the hash for multiple combining characters presented in different orders will be the same. This is because typically such hashes implement some transformation on the existing hash, together with adding the new character to the hash being accumulated. Such methods of hash construction will arrive at different values if the ordering of combining characters changes.

In order to create a hash with the necessary characteristics, one can construct a separate sub-hash for composite character, consisting of one non-combining character (may be pre-composed) together with the set (possibly null) of combining characters immediately following it. Each such composed character, whether precomposed or not, will have its own sub-hash, which will be the same regardless of the order of the combining characters.

If the hash is to include case-insensitivity, special handling is needed to deal with issues arising from the handling of COMBINING GREEK YPOGEGRAMMENI (U+0345). That combining character, as discussed in item EX6 of Appendix A.1 is uppercased to the non-combining character GREEK CAPITAL LETTER IOTA (U+0399) which is in turn lowercased to the non-combining character GREEK SMALL LETTER IOTA (U+03B9). As a result, when computing a case-insensitive hash, when a base character is IOTA (of either case) and the previous base character is ALPHA, ETA, or OMEGA (of the same case as the IOTA), that IOTA is treated, for the purpose of defining the composite characters for which to generate sub-hashes as if it were a combining character. As a result, in this case a string of containing two composite characters will be treated as were a single composite character since the iota will be treated as if it were a combining character. This string will have its own sub-hash, which will be the same regardless of the order of combining characters.

The same outline will be followed for generating hashes which are to be form-insensitive (only) and for those which are to be both form-insensitive and case-insensitive. The initial value, representing the base character, will differ based on the type of hash, as discussed below.

* In the case-sensitive case, the initial value of the sub-hash will reflect the value of the base character with the only possible need to map to a different value deriving from the existence of OHM SIGN (U+2126), ANGSTROM SIGN (U+212B), and MICRO SIGN (U+00B5)

as characters distinct from the letters that represent these code points. This could be done with a mapping table but most implementations would probably choose to implement special-purpose code to do this.

* In the case-insensitive case, the initial value of the sub-hash will reflect the case-based equivalence class to which the character (the lower-case equivalent is generally suitable). In this context a table-based mapping is required and this mapping can shift OHM SIGN, ANGSTROM SIGN, and MICRO SIGN to the case-based equivalence class for the corresponding character.

Regardless of the type of hash to be produced, values based on the following combining characters need to reflected in the sub-hash. In order to make the sub-hash invariant to changes in the order of combining characters, values based on the particular combining character are combined with the hash being computed using a commutative associative operation, such as addition.

To reduce false-positives, it is desirable to make the hash relatively wide (i.e. 32-64 bits) with the value based on base character in the upper portion of the word with the values for the combining characters appearing in a wide range of bit positions in the rest of the word to limit the degree that multiple distinct sets of combining characters have value that are the same. Although the details will be affected by processor cache structure and the distribution of names processed, a table of values will be used but typical implementations will be different in the two cases we are dealing as described in Appendix B.2.2.

As each sub-hash is computed, it is combined into a name-wide hash. There is no need for this computation to be order-independent and it will probably include a circular shift of the hash computed so far to be added to the contribution of the sub-hash for the new base or composed character.

As described in Appendix B.2.3 the appropriate full name hash will have the major role in excluding potential matches efficiently. However, in some small number of cases, there will be a hash match in which the names to be compared are not equivalent, requiring more involved processing. It is assumed below that a given name will be searching for potential cached matches within the directory so that for that name, on will be able retain information used to construct the full name hash (e.g. individual sub-hashes plus the bounds of each composite character. These will be compared against cached entries where only the full (e.g. 64-bit) name hash and the name itself will be available for comparison.

B.2.2. Character Tables

The per-character tables used in these algorithms have a number of type of entries for different types of characters. In some cases, information for a given character type will be essentially the same whether the comparison is to be form-insensitive or case-insensitive. In others, there will be differences. Also, there may be entry types that only exist for particular types of comparisons. In any case, some bits within the table entry will be devoted to representing the type of character and entry, with provisions for the following cases:

* For combining characters, the entry will provide information about the character's contribution to the composite character sub-hash in which it appears.

* For case-insensitive comparisons, there needs to be special entries for characters, which, while not themselves combining characters, are the case-insensitive equivalents of combining characters. An example of this situation is provided in item EX6 within Appendix A.1.

* For pre-composed characters, the entry needs to provide the initial hash value which is to be the basis for the sub-hash for the name substring including contributions for the base character together with contribution of included combining characters. In addition, such entries will provide, separately, information about the character's canonical decomposition.

* For case-insensitive comparisons, there needs to be, for base characters, entries assigning each base character to the case-based equivalence class to which it belongs, although such entries can be avoided if the equivalence class matches the character (usually caseless and lowercase characters.

* Also, for case-insensitive comparisons, there will need to be special entries for characters which multi-character string as case-insensitive equivalent of the base character. Examples of this situation are provided in items EX4 and EX5 within Appendix A.1. Such entries will need to have a hash-contribution that reflects the hash that would be computed for the multi-character string.

* For form-insensitive comparisons, there will be special entries to provide special handling for those cases in which there are two canonically equivalent single characters. Such entries do not exist for case-insensitive comparison since this situation can be handled by a non-standard use of case mapping for base characters by placing these two characters in the same case-based equivalence

In the common case in which a two-stage mapping will be used, there will be common groups of characters in which no table entry will be required, allowing a default entry type to be used for some character groups with entry contents easily calculable from the code point.

* In the case form-insensitive comparison, this consists of all base characters, with the hash contribution of the character derivable by a pre-specified transformation of the code point value.

* In the case case-insensitive comparison, this consists of all base character which are either caseless or equivalence class is the same as the code point, typically lowercase characters. As in the form-insensitive case, the hash contribution of the character is derivable by a pre-specified transformation of the code point value, which matches, in this case, the id assigned to the case-based equivalence class.

B.2.3. Outline of comparison

We are assuming that comparisons will be based on the hash values computed as described in Appendix B.2.1, whether the comparison is to be form-insensitive or both case-insensitive and form-insensitive.

To facilitate this comparison, the name hash will be stored with the names to be compared. As a result, when there is a need to investigate a new name and whether there are existing matches, it will be possible to search for matches with existing names cached for that directory, using a hash for the new name which is computed and compared to all the existing names, with the result that the detailed comparisons described in Appendices B.2.4 and B.2.5 have to be done relatively rarely, since non-matching names together with matching hashes are likely to be atypical.

Given the above, it is a reasonable assumption, which we will take note of in the sections below, that for one of the names to be compared, we will have access to data generated in the process of computing the name hash while for the other names, such data would have to be generated anew, when necessary. When that data includes, as we expect it will, the offset and length of the string regions covered by each sub-hash, direct byte-by-byte comparisons between

corresponding regions of the two strings can exclude the possibility of difference without invoking any detailed logic to deal with the possibility of canonical equivalence or case-based equivalence in the absence of identical name segment.

In the case in which the byte-by-byte comparisons fail, further analysis is necessary:

* First, the associated base characters are compared, as is discussed in Appendix B.2.4. When doing form-insensitive comparison this is straightforward. However, when case-insensitive comparison is to be done, there is the possibility that the sub-hash boundaries of the two comparands are different, requiring that a common point in both comparands be found to resume comparison after a successful match. For either form of comparison, if a mismatch is found at this point then the comparison fails, while, if there is match, there must be a comparison of any following combining characters, as described below, before moving on to the region covered by the appropriate sub-string covered by the appropriate next sub-hash for each comparand.

* If there is no mismatch as to the base characters, the set of associated combining characters (might be null) must be compared, as is discussed in Appendix B.2.5. If a mismatch is found at this point then the comparison fails. This may be because the sets of combining characters are different, because there are multiple copies of the same combining character in one of the string, or because the difference in combining character is not one that maintains canonical equivalence (due to combining classes).

* When both comparisons show a match, the comparison resumes at the next substring, using a byte-by-byte comparison initially. If the comparison cannot be resumed because one of the strings is exhausted, the comparison terminate, succeeding only if both strings are exhausted while failing if only one of the strings is exhausted.

B.2.4. Comparing Base Characters

In general, the task of comparing based characters is simple, using a table lookup using the numeric value of the initial character in the substring. When doing form-insensitive comparison this is the base character associated with the initial (possibly pre-composed) character, while for case-insensitive comparison it is the case-based equivalence class associated with that character.

When doing case-insensitive comparison, issues may arise that result when there is a multi-character string that as the case- insensitive equivalent of a single base character, as discussed in items EX4 and EX5 within Appendix A.1. These are best dealt with using the approach outlined in Appendix B.1. When it is noted that the current base character (for either comparand) is a character whose associated equivalence class contains one or more multi-character strings, then these comparisons, normally requiring that each base character be mapped to the same case-based equivalence class be modified to allow equivalences allowed by these multi-character sequences.

In such cases, there may need to be comparisons involving the multi-character string, in addition to the normal comparisons using the base characters' equivalence class. As an illustration, we will consider possible comparison results that involve characters string within the equivalence class mentioned in item EX4 within Appendix A.1.

* When the base character for both comparands are either LATIN SMALL LETTER SHARP S (U+00DF) or LATIN CAPITAL LETTER SHARP S (U+1E9E), then a match is recognized.

* When the base character for one comparand is either LATIN SMALL LETTER SHARP S (U+00DF) or LATIN CAPITAL LETTER SHARP S (U+1E9E), while the other is not, each character in the that other comparand is case-insensitively compared to the corresponding character of the string "ss" with a match being signaled when all such subsequent characters match, except for possibly being of a different case. Because that comparison will involve multiple base characters, the overall comparison point for that comparand will have to be adjusted to reflect character already processed as part of the comparison.

* When the base character for neither comparands is either LATIN SMALL LETTER SHARP S (U+00DF) or LATIN CAPITAL LETTER SHARP S (U+1E9E), then matching proceeds normally. As a result, the only cases in which character strings within the equivalence class being discussed will result is where both comparands have one of the strings "ss", "sS", "Ss", or "SS" at the current comparison point.

B.2.5. Comparing Combining Characters

In order to effect the necessary comparison, one needs to assemble, for each comparand, the set of combining characters within the current substring. The means used might be different for different comparands since there might be useful information retained from the generation of the associated string hash for one of the comparands. In any case, there are two potential sources for these characters:

* Those deriving from the canonical decomposition of a pre-composed character, treated as a null set of if the base character is not a precomposed one.

* Those combining characters that immediately follow the base character, which will be a null set if the immediately following character is not a combining character. Note that it is possible, when doing case-insensitive comparison to treat certain character, not normally combining characters, as if they are. Such situations can arise, when, as described in item EX6 within Appendix A.1, such non-combining character are the uppercase or lowercase equivalents of combining characters.

Although, the two sets of character can be checked to see if they are identical, this is a sufficient but not a necessary condition for equivalence since some permutations of a set of combining characters are considered canonically equivalent. To summarize the appropriate equivalence rules:

* Combining characters of different combining classes may be freely reordered.

* If combining characters of the same combining class are reordered, then result is not canonically equivalent

The rules above do not directly apply to the case, discussed above, in which some non-combining characters are the case-based equivalents of combining characters such as COMBINING GREEK YPOGEGRAMMENI (U+0345). Nevertheless, because of this equivalence, those implementing case-insensitive comparisons do have to deal with this potential equivalence when considering whether two strings containing combining characters or their case-based equivalents match. As a result when comparing strings of combining characters, we need to implement the following modified rules.

* When one comparand has a true combining character and the other comparand has an identical one, they may differ in location as long as there is no permutation of combining characters of the same combining class.

* When one comparand has a true combining character and the other has a case-insensitive equivalent which is not a combining character, that character must appear last in its string while the combining may character appear in its string in any position except the last. In this case, there are no restrictions based on combining classes.

* When both comparands contain a non-combining character case-insensitively equivalent to a combining character, these character must appear last in their respective strings.

Although it is possible to divide combining characters based on their combining classes, sort each of the list and compare, that approach will not be discussed here. Even though the use of sorts might allow use of an overall N log N algorithm, the number of combining characters is likely to be too low for this to be a practical benefit. Instead, we present below an order N-squared algorithm based on searches.

In this algorithm, one string, chosen arbitrarily, is designated the "source string" and successive characters from it, are searched for in the other, designated the "target string". Associated with the target string is a mask to allow characters search for a found to be marked so that they will not be found a second time. In the treatment below, when a character is "searched for" only characters not yet in the mask are examined and the character sought has its associated mask bit set when it is found.

Each character in the source string is processed in turn with the actual processing depending on particular character being processed, with the following three possibilities to be dealt with.

1. For the typical case (i.e. a combining character with no case-insensitive equivalents), the character is searched for in the target string with the compare failing if it is not found.

   If it is found, then the region of the target string between the point corresponding to the current position in the source string and the character found is examined to check for characters of the same combining class. If any are found, the overall comparison fails.

2. For the case of a combining character with a case- insensitive equivalents, the character is searched for as described in the first paragraph of item 1. However, the compare does not fail if it is not found. Instead, a case-insensitive equivalent character is searched for at the final position of the string and the compare fails if that is not found.

3. For the case of a non-combining character that has a combining character as a case-insensitive equivalents, the overall comparison fails if the character is not in the final position within the source string or has already been successfully searched for. Otherwise, the corresponding combining character is searched for in the target as described in in the first paragraph of item 1. The overall compare fails if it is not found.

Once all characters in the source string has been processed, the mask associated is examined to see if there are combining character that were not found in the matching process described above. Normally, if there are such characters, the overall comparison fails. However, if the last character of the target was not matched and if it is a non-combining character that is case-insensitively equivalent to a combining character, then comparison succeeds and the remaining character needs to be matched with the next substring in the source.

B.3. Optimization of Form-Insensitive Comparisons

This section will discuss situations in which form-independent comparisons, for certain groups of strings, can be done in a more efficient manner than described in Appendix B.2.

One important group of strings is those in which all of the characters consist of a single byte. We call these strings the UTF8-onebyte subset. A string’s membership in this subset can be easily determined as part of UTF8-compliance checking, hash generation, or a preliminary byte-by-byte comparison to a string whose membership status in this subset is already known.

As a result, there are many situations in which a form-independent string comparison can be done without reference to detailed character tables or any UTF8-to-UCS conversions. Examples follow:

* If the current file system is case-sensitive and either of two strings being compared are a member of the UTF8-onebyte subset the result of a byte-by-byte comparison of the two strings can be accepted as definitive without any reference to the details of the particular canonical equivalence relation used.

  When neither of the strings being compared are a member of the UTF8-onebyte subset, there are further opportunities for optimized comparisons, discussed below.

  This applies regardless of the particular Unicode version used.

* If the current file system is case-insensitive and the handling of case equivalence is such that LATIN SMALL LETTER I (U+0069), and LATIN CAPITAL LETTER I (U+0049) are considered equivalent, then, when both of the strings being compared are members of UTF8-onebyte subset, a positive result for the comparison can be immediately accepted but a negative result, need to be supplemented by simple version of case-insensitive comparison using a 127-byte table mapping each letter to other-case equivalent. If this succeeds the strings are equivalent, while, if it does not, all the complexities of form-insensitive string comparisons need to be taken account of.

  This applies regardless of the particular Unicode version used.

* If the current file system is case-insensitive and the handling of case equivalence is such that either LATIN SMALL LETTER I (U+0069), and LATIN CAPITAL LETTER I (U+0049) are not considered equivalent, or the handling of these characters is unknown (client only) than a variant of the above can be used.

  In this variant, when a byte-by-byte comparison results in a negative result, a byte-by-byte comparison still needs to be done but the mapping table used is different in that it does not map LATIN SMALL LETTER I (U+0069) and LATIN CAPITAL LETTER I (U+0049) to each other but maps each character to itself as it does for characters that have no case.

When the procedures above are not usable, further opportunities for optimized handling depend on case-sensitivity. For case-sensitive file systems, there are optimized approaches to name comparisons that can be used when either or both of the names being compared is not a member of the UTF8-onebyte subset.

The alternative allows a byte-by-byte comparison to be used for name comparison if at least one of the names belong to the canonical-singleton subset of strings, defined as those strings that are known to have no canonically equivalent strings. Two important facts, which implementations can take advantage of, are the following:

* The UTF8-onebyte subset is contained within the canonical-singleton subset.

  This fact can be taken advantage of when one of the two string to be compared is a member of the UTF8-onebyte subset, so no further checking is necessary in this case. As a result additional testing for membership in the canonical-singleton subset only needs to be done when neither of the two strings is a member of the UTF8-onebyte subset.

* This set can be usefully defined without reference to the particular version of Unicode to be used. This allows this set to be used by clients in testing names for suitability for negative name caching, as described in Appendix B.4.

  The set of characters can be defined as all the characters defined in a relatively early version of Unicode with certain exclusions, excluding characters which are the NFC form of some string, combining characters, defined as those ever present within some NFD form of a one-character string, together with OHM SIGN (U+2126).

  This set does not have to be changed with new Unicode versions, since, while it possible for them to add new characters to this set it is impossible to remove them since that would require converting a previously-existing character to be a combining character or given it a new decomposition which is impossible.

Implementations are likely to implement a test for strings in the canonical-singleton subset, limited to strings which are limited to strings whose UTF-8 encoding includes no character requiring more than two bytes to encode. In testing for membership in this subset one-but character can be ignored and two-byte character need to checked against a 240-byte read-only bitmap whose bytes are likely to be available quite quickly in processor caches.

B.4. Restricted Client Caching to Deal with Name Equivalences

Given the name caching difficulties mentioned in Section 7.3 and the typical lack of information regarding the details many clients will want to limit name caching as described in that section. However, there might be situations in which other approaches are desirable and we discuss the issues below:

* For case-sensitive file systems, name which are in the canonical-singleton subset can effectively cached, so clients could use the full-range of name-caching techniques for such names, even the absence of detailed information about the canonical equivalence relation being used.

  There is overhead added by this check on the client, since, unlike the server case, there is no opportunity to combine this check with validation of UTF-8 encoding. Nevertheless, that overhead is quite small so it is likely that clients will implement it for UTF8-aware file system that are case-sensitive, rather than living with restricted name caching, as described in Section 7.3.

* For case-insensitive file systems, the situation is different. Even for the UTF8-onebyte subset, the possibilities of unexpected equivalence due to issues with dotted and dotless i, sharp s, and various ligatures means that simple case-based equivalences cannot be assumed.

  As a result, clients handling case-insensitive file systems are most likely to simply avoid potentially troublesome forms of name caching, unless full information on the equivalence relation is available. In the case that it is available, all forms of name caching would be possible, but that requires the implementation on the client of the comparison methods described in Appendix B.2 together with the potential optimizations discussed in Appendix B.3.

Appendix C. History

This section describes the history of internationalization within NFSv4. Despite the fact that NFSv4.0 and subsequent minor versions have differed in many ways, the actual implementations of internationalization have remained the same and internationalized names have been handled without regard to the minor version being used. This is the reason the document is able to treat internationalization for all NFSv4 minor versions together.

During the period from the publication of RFC3010 [RFC3010] until now, two different perspectives with regard to internationalization have been held and represented, to varying degrees, in specifications for NFSv4 minor versions.

* The perspective held by NFSv4 implementers treated most aspects of internationalization as basically outside the scope of what NFSv4 client and server implementers could deal with. This was because the POSIX interface treated file names as uninterpreted strings of bytes, because the file systems used by NFSv4 servers treated file names similarly, and because those file systems contained files with internationalized names using a number of different encoding methods, chosen by the users of the POSIX interface. From this perspective, wider support for internationalized names and general use of universal encodings was a matter for users and applications and not for protocol implementers or designers.

* Within the IETF in general and in the IESG, there was a feeling that new protocols, such as NFSv4, could not avoid dealing with internationalization issues, making it difficult to treat these matters, as the implementers’ perspective would have it, as essentially out of scope.

As specifications were developed, approved, and at times rewritten, this fundamental difference of approach was never fully resolved, although, with the publication of RFC7530 [RFC7530], a satisfactory modus vivendi may have been arrived at.

Although many specifications were published dealing with NFSv4 internationalization, all minor versions used the same implementation approach, even when the current specification for that minor version specified an entirely different approach. As a result, we need to treat the history of NFSv4 internationalization below as an integrated whole, rather than treating individual minor versions separately.

* The approach to internationalization specified in RFC3010 [RFC3010] sidestepped the conflict of approaches cited above by discussing the reasons that UTF-8 encoding was desirable while leaving file names as uninterpreted strings of bytes. The issue of string normalization was avoided by saying "The NFS version 4 protocol does not mandate the use of a particular normalization form at this time."

  Despite this approach’s inconsistency with general IETF expectations regarding internationalization, RFC3010 was published as a Proposed Standard. NFSv4.0 implementation related to internationalization of file names followed the same paradigm used by NFSv3, assuring interoperability with files created using that protocol, as well as with those created using local means of file creation.

* When it became necessary, because of issues with byte-range locking, to create an rfc3010bis, no change to the previously approved approach seemed indicated and the drafts submitted up until [I-D.ietf-nfsv4-rfc3010bis] closely followed RFC3010 as regards internationalization. The IESG then decided that a different approach to internationalization was required, to be based on stringprep [RFC3454] and rfc3010bis was accordingly revised, replacing all of the Internationalization section, before being published as RFC3530 [RFC3530].

  These changes required the rejection of file names that were not valid UTF-8, file names that included code points not, at the time of publication, assigned a Unicode character (e.g. capital eszett) or that were not allowed by stringprep (e.g. Zero-width joiner and non-joiner characters). Because these restrictions would have caused the set of valid file names to be different on NFS-mounted and local file systems there was no chance of them ever being implemented.

Because these specification changes were made without working group involvement, most implementers were unaware of them while those who were aware of the changes ignored them and continued to develop implementations based on the internationalization approach specified in RFC3010.

* When NFsv4.1 was being developed, it seemed that no changes in internationalization would be needed. Many working group participants were unaware of the stringprep-based requirements which made the NFSv4.0 internationalization specified in RFC3530 unimplementable. As a result, the internationalization specified in RFC5661 [RFC5661] was based on that in RFC3530 [RFC3530], although the addition of the attribute fs_charset_cap, discussed below, provided additional flexibility.

  The attribute fs_charset_cap, discussed below in Section 9 provides flags allowing the server to indicate that it accepts and processes non-UTF-8 file names. Rejecting them was a "MUST" in RFC3530 and became a "SHOULD" in RFC5661, although there is no evidence that any of these designations ever affected server behavior.

  Even though NFSv4.1 was a separate protocol and could have had a different approach to internationalization, for a considerable time, the internationalization specification for both protocols was based on stringprep (in RFC3530 and RFC5661) while the actual implementations of the two minor versions both followed the approach specified in RFC3010, despite its obsoleted status. This happened since most working group members were aware of the treatment internationalization by the various minor version RFCs.

* When work started on rfc3530bis it was clear that issues related to internationalization had to be addressed. When the implications of the stringprep references in RFC3530 were discussed with implementers it became clear that mandating that NFSv4.0 file names conform to stringprep was not appropriate. While some working group members articulated the view that, because of the need to maintain compatibility with the POSIX interface and existing file systems, internationalization for NFSv4 could not be successfully addressed by the IETF, the rfc3530bis draft submitted to the IESG did not explicitly embrace the implementers’ perspective as set forth above.

  The draft submitted to the IESG and RFC7530 [RFC7530] as published provided an explanation (see Section 5) as to why restrictions on character encodings were not viable. It allowed non-UTF-8 encodings to be used for internationalized file names while defining UTF-8 as the preferred encoding and allowing servers to

reject non-UTF-8 string as invalid. Other stringprep-based string restrictions were eliminated. With regard to normalization, it continued to defer the matter, leaving open the possibility that one might be chosen later.

This approach is compatible, in implementation terms, with that specified in the obsolete document RFC3010 [RFC3010], allowing it to be used compatibly with existing implementations for all existing minor versions. This is despite the fact that RFC8881 [RFC8881] specifies an entirely different approach.

As a result of discussions leading up to the publishing of RFC7530, it was discovered that some local file systems used with NFSv4 were configured to be both normalization-aware and normalization-preserving, mapping all canonically equivalent file names to the same file while preserving the form actually used to create the file, of whatever form, normalized or not. This behavior, which is legal according to RFC3010, which says little about name mapping is probably illegal according to stringprep. Nevertheless, it was expressly pointed out in RFC7530 as a valid choice to deal with normalization issues, since it allows normalization-aware processing without the difficulties that arise in imposing a particular normalization form, as described in Section 7.1.

In its discussion of internationalized domain names, RFC7530 [RFC7530] adopted an approach compatible with IDNA2003, rather than attempting to derive the specification from the behavior of existing implementations.

* When IDNA2003 was replaced by IDNA2008, the internationalization specified by [RFC7530] was not changed. Also, it appears unlikely that implementations were changed to reflect that shift.

* NFSv4.2 made no changes to internationalization. As a result, RFC7862 [RFC7862] which made no mention of internationalization, implicitly aligned internationalization in NFSv4.2 with that in NFSv4.1, as specified by RFC5661 [RFC5661].

  As a result of this implicit alignment, there is no need for this document to specifically address NFSv4.2 or be marked as updating RFC7862. It is sufficient that it updates RFC8881, which specifies the internationalization for NFSv4.1, inherited by NFSv4.2.

* Later, as work on the predecessors of this document was underway, further discussion of internationalization issues made it necessary that some gaps in the discussion of internationalization

in [RFC7530] be filled in. These gaps primarily concerned the need for NFSv4 clients to match the handling of the corresponding server when using cached file name data locally, or to avoid making invalid assumptions about that handling, when information on the details of such handling was not available.

The above history, can, for the purposes of the rest of this document be summarized in the following statements:

* The actual treatment of internationalization within NFSv4 has not been affected by the particular minor version used, despite the fact that the specifications for the minor versions have often differed in their treatment of internationalization.

* With regard to file names, most implementations have followed the internationalization approach specified in RFC3010, which is compatible with the treatment in RFC7530.

* With regard to internationalized domain names, RFC7530 [RFC7530] specified an approach compatible with IDNA at the time of publication. However, no detailed analysis was done to determine whether NFSv4 implementations actually followed that approach and it appears that many implementations used approaches that were much simpler.

* Because [RFC7530] did not specifically address the special issues that clients would face, relying on the assumption that each file is accessible only by its name. As this assumption is no longer true when internationalized name handling is in effect, the appropriate handling is discusssed below. Section 7.3 explains the options for handling in the case in which the client has very limited information about the details about the server's internationalization-related handling of file names while Appendices A.3 A.4 discuss how a client might use more complete information provided by new attributes.

In order to deal with all NFSv4 minor versions, this document follows the internationalization approach defined in RFC7530, with some changes discussed in Section 4 and applies that approach to all NFSv4 minor versions.

Appendix D. Future Minor Versions and Extensions

As presented in the document proper, all current NFSv4 minor versions allow use of arbitrary string encodings, allow servers a choice of whether to be aware of normalization issues or not, and allow servers a number of choices about how to address normalization issues. This range of choices reflects the need to accommodate existing file systems and user expectations about character handling which in turn reflect the assumptions of the POSIX model for the handling file names.

While it is theoretically possible for a subsequent minor version to change these aspects of the protocol (see [RFC8178]), this section will explain why any such change is highly unlikely, making it expected that these aspects of NFSv4 internationalization handling will be retained indefinitely. As a result, any new minor version specification document that made such a change would have to be marked as updating or obsoleting this document

No such change could be done as an extension to an existing minor version or in a new minor version consisting only of OPTIONAL features. Such a change could only be done in a new minor version, which, like minor version one, was prepared to be incompatible to some degree with the previous minor versions. While it appears unlikely that such minor versions will be adopted, the possibility cannot be excluded, so we need to explore the difficulties of changing the aspects of internationalization handling mentioned above.

* Establishing UTF-8 as the sole means of encoding for internationalized characters, would make inaccessible existing files stored with other encodings. Further, unless there were a corresponding change in the UNIX file interface model, it would cause the set of valid names for local and remote files to diverge.

* Imposing a particular normalization form, in the sense of refusing to create to allow access to files whose UTF-8-encoded names are not of the selected normalization form would give rise to similar difficulties.

* Defining a preferred normalization form to be returned as the names of all internationalized files, would result in applications having to deal with sudden unexplained changes of file names for existing files.

None of the above appears likely since there does not seem to be any corresponding benefits to justify the difficulties that adopting them would create.

There would also be difficulties in otherwise reducing the set of three acceptable normalization handling options, without reducing it to a single option by imposing a specific normalization form.

* Eliminating the possibility of a single possible normalization form, would pose similar difficulties to imposing the other one, even if representation-independent comparisons were also allowed.

  In either case, a specific normalization form would be disfavored, with no corresponding benefit.

* Allowing only representation-independent lookups would not impose difficulties for clients, but there are reasons to doubt it could be universally implemented, since such name comparisons would have to be done within the file system itself.

  Such a change could only be made once file system support for representation-independent file lookups would become commonly available. As long as the POSIX file naming model continues its sway, that would be unlikely to happen.

One possible internationalization-related extension that the working could adopt would be definition of OPTIONAL per-fs attributes defining the internationalization-related handling for that file system. That would allow clients to be aware of server choices in this area and could be adopted without disrupting existing clients and servers. Appendices A.3 and A.4 discuss the possible forms of such attributes.

Acknowledgements

This document is based, in large part, on Section 12 of [RFC7530] and all the people who contributed to that work, have helped make this document possible, including David Black, Peter Staubach, Nico Williams, Mike Eisler, Trond Myklebust, James Lentini, Mike Kupfer and Peter Saint-Andre.

The author wishes to thank Tom Haynes for his timely suggestion to pursue the task of dealing with internationalization on an NFSv4-wide basis.

The author wishes to thank Nico Williams for his insights regarding the need for clients implementing file access protocols to be aware of the details of the server's internationalization-related name processing, particularly when case-insensitive file systems are being accessed.

The author wishes to thank Christoph Helwig for his insightful comments regarding the implementation constraints that internationalization-aware servers have to deal with to support normalization and case-insensitivity.

Author's Address

David Noveck
NetApp
201 Jones Road
Waltham, MA 02451
United States of America
Phone: +1 781 572 8038
Email: davenoveck@gmail.com