Workgroup: NFSv4
Internet-Draft:
draft-ietf-nfsv4-internationalization-07
Updates: 8881, 7530 (if approved)
Published: 20 November 2023
Intended Status: Standards Track
Expires: 23 May 2024
Authors: D. Noveck
NetApp

Internationalization for the NFSv4 Protocols

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 23 May 2024.

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 A. 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 physical file system. It is common for servers and physical file systems to be configurable as to the behavior shown. In the discussion below, each configuration that shows different behavior is 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 the specfications 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 essentially 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 A. For a discussion of those differences and potential compatibility issues, see Sections 12.1 and 12.2.

*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 specification 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.

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 beyound NFSv4.1 (i.e. to NFSv4.2), which made no changes in the specification of internationalization-related handling and maintained existing implementation patterns,

2. Requirements Language

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. Requirements Language Derivation

Although the key words "**MUST**", "**SHOULD**", and "**MAY**" retain their normal meanings, as described above, we need to explain how the statements involving these terms were arrived at:

*In the case of statements within Sections 12 and 15, these derive from the requirements of other internet specifications.

*In the case of statements within Sections 7, 10, and 11 derive from the author's view of the appropriate normative language to use and will, when this document is advanced, represent the working group's consensus on those same matters.

* However, in other cases, i.e. those in sections deriving from RFC7530 [RFC7530] (i.e. Sections 5, 6, 8, 9, 13, 14, 16, 17) this specification's descriptions were derived from existing implementation patterns. Although this pattern is atypical, it is needed to provide a description that satisfies the goal of [RFC2119], providing a normative description to enable future implementations to be compatible with existing ones. This requires that we explain later in this section how the normative terms used derive from the behavior of existing implementations, in those situations in which existing implementation behavior patterns can be determined.

Note that in introductory and explanatory sections of this document (i.e. Sections 1 through 4 these terms do not appear except to explain how they are used in this document. Also, they do not appear in Appendix B which provides non-normative implementation guidance.

With regard to the parts of this document deriving from RFC7530, we explain below how the normative terms used derive from the behavior of existing implementations, in those situations in which existing implementation behavior patterns can be determined.

* Behavior implemented by all existing clients or servers is described using "**MUST**", since new implementations need to follow existing ones to be assured of interoperability. While it is possible that different behavior might be workable, we have found no case where this seems reasonable.

  The converse holds for "**MUST NOT**": if a type of behavior poses interoperability problems, it **MUST NOT** be implemented by any existing clients or servers.

* Behavior implemented by most existing clients or servers, where that behavior is more desirable than any alternative, is described using "**SHOULD**", since new implementations need to follow that existing practice unless there are strong reasons to do otherwise.

  The converse holds for "**SHOULD NOT**".

* Behavior implemented by some, but not all, existing clients or servers is described using "**MAY**", indicating that new implementations have a choice as to whether they will behave in that way. Thus, new implementations will have the same flexibility that existing ones do.

* Behavior implemented by all existing clients or servers, so far as is known -- but where there remains some uncertainty as to details -- is described using "should". Such cases primarily concern details of error returns. New implementations should

follow existing practice even though such situations generally do not affect interoperability.

There are also cases in which certain server behaviors, while not known to exist, cannot be reliably determined not to exist. In part, this is a consequence of the long period of time that has elapsed since the publication of the defining specifications, resulting in a situation in which those involved in the implementation work may no longer be involved in or be aware of working group activities.

In the case of possible server behavior that is neither known to exist nor known not to exist, we use "**SHOULD NOT**" and "**MUST NOT**" as follows, and similarly for "**SHOULD**" and "**MUST**".

* In some cases, the potential behavior is not known to exist but is of such a nature that, if it were in fact implemented, interoperability difficulties would be expected and reported, giving us cause to conclude that the potential behavior is not implemented. For such behavior, we use "**MUST NOT**". Similarly, we use "**MUST**" to apply to the contrary behavior.

* In other cases, potential behavior is not known to exist but the behavior, while undesirable, is not of such a nature that we are able to draw any conclusions about its potential existence. In such cases, we use "**SHOULD NOT**". Similarly, we use "**SHOULD**" to apply to the contrary behavior.

In the case of a "**MAY**", "**SHOULD**", or "**SHOULD NOT**" that applies to servers, clients need to be aware that there are servers that may or may not take the specified action, and they need to be prepared for either eventuality.

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 A but of those only the implementation-based approach used by [RFC7530], resulted in a workable description while a number of attempts to specify an approach that implementors were to follow were all ignored by implementers.

It is expected that any future minor versions will follow a similar approach, even though there is nothing to prevent a future minor version from adopting 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. Issues relating to potential extensions within the framework specified in this document are dealt with in Section 15.

4. Changes Relative to RFC7530

This document follows the internationalization approach defined in RFC7530, with a number of significant necessary changes described below.

- 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 was specified in [RFC5661]. Issues relating to potential future minor versions and protocol extensions are addressed in Section 15.

- 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 12.2.

- There is more detailed 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. The discussion in Section 10 applies to both client and server, although issues relating to the client's knowledge of server behavior are dealt with in Section 11.

- There is additional material, dealing with the implications of server-side internationalization-related file name processing for clients that cache the results of READDIR's. This includes a discussion of options to deal with the current lack of detailed information about the server (in Section 11.2), and options for handling when more detailed information can be made available (in Section 11.3)."

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 physical file system, whose handling of character encoding and normalization is not specifiable by the IETF.

* Typical implementation patterns in UNIX systems 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 UTF-8-related processing can be provided by servers, as described in Section 6. These behaviors in which the server is aware of UTF-8 encoding relate to normalization and to case and are of two types:

* 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 file having an equivalent name rather than the one specified by the client. This approach is implementable within the server proper.

6. Summary of Server Behavior Types

This section describes the language-related behaviors that are vaidly manifested by NFSv4 servers together with the server-side file systems made accessible by such servers. These relate to the handling of normalization (see Section 6.1) and of case (see Section 6.2).

How these two aspects of behavior interact is described in Section 6.3 while the implications for client behavior are discussed in Section 6.4.

For both of these aspects, there are three choices:

* UTF-8-unaware file systems **MAY** be used.

  In this case, neither normalization-related nor case-related behavior is possible.

  Implementations need no special handling either in the server proper or in the file system itself.

*UTF-8-aware file systems **MAY** treat as indistinguishable file name strings that differ only as to nomalization form or only as to case.

This choice may be made independently for normalization and for case.

Implementations cannot be done in the server proper but would typically be done in the file system itself.

*UTF-8-aware file systems **MAY** modify file name strings to a corrsponding value of a particular nomalization form or of a particular case.

Implementations can be done in the server proper and do not require special character handling in the file system itself.

6.1. Nomalization-related Behavior Types

Servers, together with server-side file systems accessed using NFSv4, **MAY** reject component name strings that are not valid UTF-8. This choice leads to a number of types of valid server behavior, as outlined below. When these are combined with the valid normalization-related behaviors as described in Section 8, this leads to the valid combined behaviors outlined below.

*Servers that do not limit file component names on particular file systems to UTF-8 strings are very common and are necessary to deal with clients/applications not oriented to the use of UTF-8. Such servers ignore normalization-related issues, and there is no way for them to implement either a specfic normalization form or representation-independent lookups. These are best described as behaving as "UTF-8-unaware servers" for such file systems, since they treat file component names as uninterpreted strings of bytes and have no knowledge of the characters represented. See Section 13 for details.

*Servers that limit file component names within a given file system to UTF-8 strings exist with normalization-related handling as described in Section 8. These are best described as behaving as "UTF-8-only servers".

*It is possible for a server to allow component names that are not valid UTF-8, while still being aware of the structure of UTF-8 strings. Such servers could, in theory, implement either normalization or representation-independent lookups but apply those techniques only to valid UTF-8 strings. Such servers are not common, but it is possible to configure at least one known server to have this behavior. This behavior **SHOULD NOT** be used, due to the possibility that a file name using one encoding may,

by coincidence, have the appearance of a UTF-8 file name; the results of UTF-8 normalization or representation-independent lookups are unlikely to be correct in all cases, when considered from the viewpoint of the other encoding. Such difficulties can be compounded when case-insensitive name handling is in effect.

To summarize, the following are the valid behaviors for particular server/file system combinations:

* Can behave without knowledge of UTF-8-based internationalization, and behave as UTF8-unaware servers for some file systems.

  This is a common implementation pattern and is compatible with the expectations of those using NFSv3.

* Enforce use of names that are valid UTF-8 strings, without manifesting any normalization-aware behavior. Such servers are UTF-8-aware but treat different normalization forms of the same name as different file names.

  This behavior pattern is compatible with existing server and file system implementations but might encounter difficulties with appplications dealing with internationalization using encodings other than UTF-8.

* Enforce use of names that are valid UTF-8 strings and implement representaion-independent name matching in order to treat canonically equivalent strings as designating the same file.

  This behavior cannot, as a practical matter, be implemented within the server. Implementation requires implementing name matching changes within the file system. See Appendix B for helpful implementation guidance.

* Enforce use of names that are valid UTF-8 strings and deal with normalization by normalizing all names to a common normalization form.

  This behavior can be implemented within the server proper but might prove disconcerting to applications that create a file with one name and encounter it with another, canonically equivalent name.

6.2. Case-related Behavior Types

The following are the valid behaviors related to case for particular server/file system combinations:

* Can behave without knowledge of case mappings, i.e. act as a case-sensitive file system. All UTF-8-unaware file systems are

case-sensitive, although some UTF-8-aware file systems can be case-insensitive as well.

This is a common implementation pattern and is compatible with the expectations of those using NFSv3.

*Implement case-insensitive name matching in order to treat strings that only differ as to case as designating the same name.

This behavior cannot, as a practical matter, be implemented within the server. Implementation requires name matching changes within the file system.

It is often the case that the case conversion of specfic characters is not as straightforward as it might be expected to be. See Section 10 for further dicussion.

*Implement case-insensitive naming by mapping all file names to a common case, either upper or lower.

This behavior can be implemented within the server proper but might prove disconcerting to applications that create a file with one name and encounter it with a dufferent name, with unexpected case modifications.

6.3. Interaction of Normalization and Case

For the most part the behavior choices are as discussed in Sections 6.1 and 6.2. However the following issues need to be noted:

*UTF-8-unaware filesystems cannot implement normalization-related behavior. Similarly, they cannot implement case-related behavior and so are case-sensitive and case-preserving.

The reason for both of these restrictions is that a UTF-8-unaware filesysstem is inherently unaware of the character encoding used and so cannot modify characters specfied by the client it to refect normalization behvior or case.

*UTF-8-aware filessystems may choose to implement normalization handling or case handling independently.

When representation-independent name matching is combined with a similar handling of case (i.e. case-insensitive case-preserving handling), the matching of string name reflects the the fact that two canonically equivalent names are treated as identical and similarly for names differing only as to case. Implementation issues for representation-independent name matching are discussed in Appendix B.

*When case-related behavior is in effect, character-sequences that differ only in case are treated as identical. In many cases, this identification is straightforward.

  Generallly, the case mapping of Roman, Cyrillic, and Greek alphabets are well-understood and the addition of combining characters to these mappings pose no additional issues.

  There are cases, such as those involving the handling of dotted and dotless i's in Turkish and Azeri and the German Sharp S character, where different mappings could be used, requiring that client and server do not make different assumptions See Section 6.4.

6.4. Client Adaptations

The handling of names by the server affects both the client proper and client-side applications that deal with file names. When normalization-handling and case-handling behaviors are implemented by the server file system, these affect how file names are mapped to specific files.

In the case of UTF-8-unaware file systems, each distinct name string is mapped to a single corresponding file object. This applies even if two strings are canonically equivalent or differ only in the case of the corresponding string characters.

When behaviors related to normalization or case are in effect, there are two possibilities which may be chosen for each behavior class:

*When canonically equivalent names, or names which only differ as to case, are considered the same, there can only be a single such file, that share the equivalent name.

  In such situations, users normally expect such name identiication to occur.

*When specified file names are normalized to a different form, or converted to a different case, the file name will differ from that specified by the user.

  In such situations, users are typically prepared for this sort of name variance.

When clients cache the existence or non-existence of files with particuar names, as they often do, they need to take account of servers handling of normalization and case.

This issue rarely affects cases of positive name caching in which the client records the existence of a file with a particular name.

Clients have to avoid assuming that the name create is that povided by the client, shich is nit universally true

In the case of negative name caching, clients might record the non-existence of a file with a particular name and be unaware that the creation of a file with a different, although equivalent, name by another client. To prevent that, the client would have to forego negative name caching or have a way of determinine the server's notion of name equivalence. With such information, the client can adapt its notion of name equivalence to be that used by the server. See Section 11 for details.

7. The Attribute Fs_charset_cap

This attribute, nominally "RECOMMENDED", 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 problems noted in Section 7.1, the overall scheme was never implemented and NFSv4.1 implementations dealt with internationalization in the same way as NFSv4.0 implementations had.

It is generally accepted that NFSv4 attributes designated elsewhere as "RECOMMENDED" are essentially **OPTIONAL** with the client having the responsibility to deal with server non-support of them. While RFC7530 has gone so far as to explicitly exclude this use from the general statement that these terms are to be used as defined by RFC2119, no NFSv4.1 specification has done so, at least through [RFC8881]. In this particular case, there are a number of circumstances that makes this **OPTIONAL** status noteworthy:

* The statement "It is expected that servers will support all attributes they comfortably can and only fail to support attributes that are difficult to support in their operating environments", appearing in Section 5.2 of [RFC8881] is troublesome since it is hard to understand how a server could find this read-only attribute "difficult to support" regardless of the operating environment used.

* This was added in minor version one which added a number of **REQUIRED** operations and could well have added a **REQUIRED** attribute.

* The fact that the client is to be prepared for non-support of the attribute would require specification of a default value to be assumed by the client, yet none is provided.

The attribute contains two flag bits. As discussed below, in Section 7.1, it is hard two see why two bits are required while the implications of this issue for future NFSv4.1 specifications will be discussed in Section 7.2

7.1. The Attribute Fs_charset_cap in Published NFSv4.1 Specifications

We reproduce Section 14.4 of [RFC8881] below, with comments interspersed trying to make sense of what is there, in order to arrive at an appropriate replacement, to be presented in Section 7.2. In that connection, we need to understand better a few issues:

* The use of two bits while one is clearly adequate, given the subject matter actually mentioned.

* The mention of possible "capabilities" which could not possibly be realized.

* The use of the RFC2119 keyword "**SHOULD**" in contexts in which this term is clearly inappropriate.

Issues related to the confusion caused by mention of "UTF-8 characters" and the lack of mention of Unicode will be addressed in the revision in Section 7.2 but will not be further discussed here.

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

typedef uint32_t        fs_charset_cap4;
```

While it is made clear that two separate bits are to be provided, their names seem to indicate that they should be complements of one another. As a way of understanding why two bits were specified, it is helpful to consider a possible boolean attribute as a potential replacement. That attribute would clearly govern whether names that do not conform to the rules of UTF-8 are to be rejected, which was a "**MUST**" in [RFC3530]. Although conveying this information is clearly part of the motivation, stating so explicitly might have been judged by the authors as unnecessarily provocative, given the role of IESG in arriving at the internationalization approach specified in RFC3530.

> Because some operating environments and file systems do not enforce character set encodings,

It is clear that the ability of operating environments to enforce use of UTF-8 encoding is not an issue, since RFC3530 made this the responsibility of the server implementation. That mandate was never followed because implementers chose not to follow it, and not because they were unable to do so.

The apparently confused statement above is best understood if one notes that its essential job is to state that the "**MUST**" in RFC3530 referred to above is not reasonable. However, the authors might well have felt unable to say so explicitly, in light of the potential IESG reaction.

> NFSv4.1 supports the fs_charset_cap attribute (Section 5.8.2.11) that indicates to the client a file system's UTF-8 capabilities.

The problem with the mention of (plural) capabilities is that the only capability mentioned which servers could implement is to accept strings which are not valid UTF-8. There are other potential capabilities having to do with the handling of canonically equivalent file nsmes, but since they were not mentioned, they will not be discussed further here.

> The attribute is an integer containing a pair of flags. The first flag is FSCHARSET_CAP4_CONTAINS_NON_UTF8, which, if set to one, tells the client that the file system contains non-UTF-8 characters,

As stated, this would mean that a server would have to keep track of a count of non-UTF-8-encoded names within the file system and change the attribute value as that count varied between zero and non-zero. Since it is most unlikely that any server would keep track of that or that any client would find it useful, we will assume that the capability to store such names is what is most likely intended.

> and the server will not convert non-UTF characters to UTF-8 if the client reads a symbolic link or directory,

There is no way for the server to convert non-UTF names to UTF-8 or anything else, since it has no knowledge of the name encoding to begin with. The alternative to treating names as UTF-8-encoded Unicode strings is to treat them as POSIX does, as uninterpreted strings of bytes. That makes it impossible to interpret strings that do not follow the rules of UTF-8 at all, making it impossible to convert the string to UTF-8.

> neither will operations with component names or pathnames in the arguments convert the strings to UTF-8.

As stated above, there is no way a server could ever do that.

> The second flag is FSCHARSET_CAP4_ALLOWS_ONLY_UTF8, which, if set to one, indicates that the server will accept (and generate) only UTF-8 characters on the file system.

That is clear and so it poses no problem for a revised treatment, unlike the other flag.

> If FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 is set to one, FSCHARSET_CAP4_CONTAINS_NON_UTF8 **MUST** be set to zero.

There is no problem with this statement. However, it does, by implication, raise the issue of what values of FSCHARSET_CAP4_CONTAINS_NON_UTF8 may be set in the case in which FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 is set to zero.

> FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 **SHOULD** always be set to one.

According to xref target="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 weighing a different course". In this context, it is unclear what these "full implications" might be, given the introduction above. The clause, "because some operating environments and file systems do not enforce character set encodings", gives one no basis for treating this as other than an unproblematic behavioral variant, calling into question the use of "**SHOULD**".

Also, the statement in RFC2119 that these terms (i.e. those like "**SHOULD**") "only be used where it is actually required for interoperation or to limit behavior which has the potential for causing harm" raises the following issues:

- The whole purpose of this feature is to enable interoperation and there is no basis for the implication that one particular flag value is superior to another in allowing interoperation.

- There is no basis for assuming that accepting file names that are not UTF-8-encoded Unicode has any potential for causing harm.

Despite the statement in RFC2119, that "they [i.e. terms such as '**SHOULD**'] must not be used to impose a particular method on implementors", it is hard to avoid the conclusion that this is in fact the motivation for the "**SHOULD**". The authors might not have had any such intention but it is posible they felt that the IESG might well have had such an intention and used "**SHOULD**" for this reason.

7.2. The Attribute Fs_charset_cap in Future NFSv4.1 Specifications

We provide a revised version of Section 14.4 of [RFC8881] below, taking into account the issues noted in Section 7.1. Given there was a working group consensus to adopt the confusing language discussed there, we must now adopt, by consensus, a clearer replacement that reflects the working group's intentions and is compatible with existing implementations. Given the passage of time and the changed context, it might not be possible to determine those intentions. In any case, we will have to be aware of how this attribute was implemented and used, particularly with regard to the first flag, whose meaning remains obscure.

The following treatment is proposed as a basis for discussion, with the understanding that it would need to be changed, if it could raise interoperability issues.

```
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 UTF-8 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 UTF-8 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 adherence to the rules of UTF-8. While such file systems are generally UTF-8-unaware, this cannot be assumed, since server are allowed (in some circumstances; it is a "**SHOULD NOT**") to accept non-UTF-8 names while being aware of the structure of UTF-8-conforming names, for the purposes of determining canonical equivalence, for example. See Section 6.

With regard to the flag FSCHARSET_CAP4_CONTAINS_NON_UTF8, it has proved impossible to determine, from existing treatments of this attribute, any value that might be helpful here. As a result, we are forced to assume that this flag is always a complement of FSCHARSET_CAP4_ALLOWS_ONLY_UTF8 and that any result in which it is not so is to be ignored, with the appropriate handling being the same as would apply if the attribute were not supported.

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.

8. 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 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 may be impossible for the NFSv4 client to enforce the use of UTF-8. The use of non-UTF-8 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 9) 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 Punycoe, and additional string format restrictions will apply. See Section 12 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.

9. Normalization

The client and server operating environments can potentially differ in their policies and operational methods with respect to character normalization (see [UNICODE] for a discussion of normalization forms). This difference may also exist between multile applications on the same client. This adds to the difficulty of providing a single normalization policy for the protocol that allows for maximal interoperability. This issue is similar to issues related to character case, in which the server may or may not support case-insensitive file name matching and may or may not preserve the character case when storing file names. The protocol does not mandate a particular behavior but allows for a range of useful behaviors.

The NFSv4 protocol does not mandate the use of a particular normalization form. A subsequent minor version of the NFSv4 protocol might specify a particular normalization form, although there would be difficulties in doing so (see Section 15 for details). In any case, the server and client can expect that they might receive unnormalized strings within protocol requests and responses. If the operating environment requires normalization, then the implementation will need to normalize the various UTF-8 encoded strings within the protocol before presenting the information to an application (at the client) or local file system (at the server).

Server implementations **MAY** normalize file names to conform to a particular normalization form before using the resulting string when looking up or creating files. Servers **MAY** also perform normalization-insensitive string comparisons without modifying the names to match a particular normalization form. Except in cases in which component names are excluded from normalization-related handling because they are not valid UTF-8 strings, a server **MUST** make the same choice (as to whether to normalize or not, the target form of normalization, and whether to do normalization-insensitive string comparisons) in the same way for all accesses to a particular file system. Servers **SHOULD NOT** reject a file name because it does not conform to a particular normalization form, as this would deny access to clients that use a different normalization form or clients acting on behalf of applications that use a different normalization form.

10. Case-Insensitive Processing of File Names

When the server is to process file names in a case-insensitive way in a given file system, it may choose to do so in a number of ways.

- It can force all characters which have multiple forms to a common case, whether upper case or lower case. Although this may cause the file name shown in the directory to be different from that specified when the file is created, these two names will be judged as equivalent when a case-insensitive comparison is used. Such file systems are case-insensitive but not case-preserving.

- It can preserve all names, presented as valid and not subject to case-based modification, while treating two names that are equivalent when a case-insensitive comparison is used as referring to the same file. Such file systems are both case-insensitive and case-preserving.

When a server implements case-insensitive file name handling, it is necessary 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 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 will 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 relavant 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 models.

* Scanning of names for multi-character sequences might well be too complicated, 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, because of the invertibility of case mappings when used to determine case-insensitive string equivalence for very large sets of strings. For example, a string of eight copies of the letter S would give rise to an 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 descibed below.

* Case-insensitive servers **MAY** do either case-mapping to a chosen 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 second of these is not allowed, unless there is a means of informing the client that it has been chosen.

*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.

10.1. 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 equivalence classes contain multi-character strings, there are potential complexities that derive from the need to recognize such multi-character strings within the inputs bein compares.

The algorithm presented in this section requires that if there is more than one multi-character string within a given equivalence class, they must all be equivalent, with any equivalences derivable from case-insensitive string equivalence using single-character equivalence classes.

Although other sources are possible (see items EX2 and EX3 in Section 10.2), multi-character sequences often appear in case-insensitive equivalence classes as the result of the canonical decomposition of one or more precomposed characters as elements of a case-insensitive equivalence class.

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

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

*For each possible character value, the associated equivalence class for case-insensitive comparison will be identified

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

* In the case of equivalence classes 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 as an equivalence class identifier.

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.

10.2. Important Examples of 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.

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 equivalence classes 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 an equivalence class 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 titlecase character LATIN CAPTAL LETTER D WITH SMALL LETTER Z (U+01F2) in addition to the uppercase LATIN CAPITAL LETTER DZ (U+01F1). While the titlecased variant 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 the same equivalence class.

This equivalence class can be derived from only C-type mappings. The possibility of mapping these characters to 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 equivalence class consisting of OHM SIGN (U+2126), GREEK CAPITAL LETTER OMEGA (U+03A9), and GREEK SMALL LETTER OMEGA (U+03C9).

This equivalence class can be derived only from 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 an equivalence class 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 equivalence class can be derived from 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, German-speakers 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 Section 10.1, as well by code specifically written to deal with this particular issue.

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 equivalence class 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 equivalence class 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 nother. 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 equivalence class, 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).

11. Internationalization-related Processing of File Names by Clients

Given the way that internationalization is addressed within the NFSv4 protocols, clients and applications accessing NFS files can generally remain unaware of the specific type of internationalization-related processing implemented by the server. For example, although a server **MAY** store all file names according to the rules appropriate to a particular normalization form, it **MUST NOT** reject names solely because they are not encoded using this normalization form, allowing the clients and applications to avoid knowledge of normalization choices.

However, as has been pointed out in [I-D.williams-filesystem-18n], there are situations in which clients implementing local optimizations use the saved contents of directories fetched from the server, making it necessary that the client's and the server's handling of internationalization-related name mapping issues be in concord. There are two basic ways this issue can be addressed:

- Where the protocol has not defined a means whereby the client can obtain information about the details of internationalized name handling implemented within the server, the client can avoid conflict with the server by limiting its use of local optimizations. While positive name caching can be used without adverse effects, negative name caching has to be limited to avoid situations in which a given name is not present but an equivalent one may exist, as far as the server is concerned. This situation, which applies to all current NFSv4 protocols is discussed in Section 11.2.

- The client could be provided complete information about the server's internationalization-related name handling (typically implemented within the server-based file system. This sort of information, which could be implemented in later NFSv4 minor versions, or in an extension to an existing extensible minor version, is discussed in Section 11.3.

- Note that when case-insensitive handling of file names is implemented by a server-side file system, further complications can arise. For the most part, these are addressed in Sections 11.2 and 11.3 by treating the particulars of case-handling as another element of the name handling implemented by the server. However, some of the specific complexities are addressed separately in Section 10.

11.1. Server Restrictions to Deal with Lack of Client Knowledge

There are a number of restrictions, not previously specified in [RFC7530], on server implementation of internationalized file name handling. These restrictions apply to both case-sensitive and case-insensitive file systems and are designed to limit the options that servers have in choosing server-side internationalized file name handling so as to enable the clients to either duplicate that handling or to limit it so as to avoid relying on cases in which the proper handling cannot be determined or duplicated by the client.

- The canonical equivalence relation implemented by the server, for each internationalization-aware file system **MUST** match that defined by some particular UNICODE version equal to or later than version 4.0.

*The case-equivalence relationship implemented by the server, for each case-insensitive file system **MUST** include all C-type case mappings included by the particular UNICODE version whose canonical equivalence relation is implemented by the server, with the possible exception of those conflicting with T-type case mappings. by some particular Unicode version equal to or later than version 4.0.

*In cases in which the server provides no way of determining the details of the case-equivalence relationship implemented by the server for a particular file system, that mapping **MUST** include all C-type case mappings included by the particular UNICODE version whose canonical equivalence relation is implemented by the server, e.g. it **MUST** map between LATIN SMALL LETTER I (U+0069)and LATIN CAPITAL LETTER I (U+0049).

11.2. Client Processing of File Names for Current NFSv4 Protocols

The existing minor versions, NFSv4.0 [RFC7530], NFSv4.1 [RFC5661], and NFSv4.2 [RFC7862], have very limited facilities allowing a client to get information about the server's internationalization-related file name handling. Because these protocols were all defined when it was assumed that the server's internationalized file name handling could be specified in great detail, there no provision was made for attributes defining the server's choices. As a result, the information available to the client is quite limited:

*The client can determine that the server is not performing internationalized file name processing. It can do this by looking up a file name using a string which is not valid UTF-8, concluding that if the LOOKUP is not rejected on that basis, then the file system is not internationalization-aware, allowing the client to ignore the potential difficulties which server-based internationalized file name processing might give rise to.

*The client can use the optional per-fs attributes case_insensitive and case_preserving to know how the server deals with character case for a particular file system. When one of these attributes is not supported by a particular file system, the client treats the attribute as if it were false.

When a file system is internationalization-unaware, the client can use both positive and negative name caching, without any issues arising from the potential for conflict between distinct file names that would be considered equivalent by the server. In other cases, the handling is more restricted in the use of negative name caching. The issue with regard to case-sensitive and case-insensitive file systems are discussed separately below. In each case, the client has a range of choices trading off the possibility of forgone

optimization opportunities against the difficulty of implementation. In doing so, it can avoid the negative consequences arising from the fact that certain details of the server's name handling are not known to it.

In the case of case-sensitive file systems, the uncertainty to be dealt with concerns the version of Unicode implemented by the server, given that different versions may have different canonical equivalence relationships. However, whether the server implements a particular normalization form or implements form-insensitive file name matching has no effect on client behavior. In light of the uncertainty created by the lack of knowledge of the precise Unicode version used by the server to implement its canonical equivalence relation, the following possibilities, arranged in order of increasing value (and difficulty of implementation) need to be considered by client implementers.

A1: The client can simply decline to implement optimizations based on negative name caching on internationalization-aware file systems.

While this might have a negative effect on performance, it might be the best option for clients not heavily used to access internationalization-aware file systems, or where, due to a lack of directory delegation support, the client has no assurance that it will be notified of the invalidation of a previous assumption that a particular file does not exist.

A2: Relatively simple name filtering can exclude the names for which negative name caching might cause difficulties. For example, the client could scan file names for characters whose presence might pose difficulties and allow negative name caching only for strings known not to contain such characters. Because the Unicode version used by the server file system is not known, this treatment would be limited to strings only containing characters defined in the earliest version of Unicode which could be supported, that is, Unicode 4.0.

One simple way for a client to provide such filtering would be to establish an upper limit (e.g. U+00FF) and disallow negative name caching for strings containing characters above that value or characters below that value that might cause there to be canonically equivalent strings on the server. A simple mask could be used to allow each character to be examined allowing composed and combining characters to be identified together with code points unassigned in Unicode 4.0.

This approach would allow negative name caching to be disallowed for strings containing those characters while allowing it for other strings that do not. A larger limit (and a corresponding mask) would make sense for clients used to access many file names containing characters from non-Latin alphabets.

A3: A client might implement its own internationalized file name handling paralleling that of the server. Because the Unicode version used by the server file system is not known, strings for which it is possible that the set of canonically equivalent strings might be different depending on the version of Unicode implemented by the server will have to be identified and excluded from using negative name caching. This would require that strings containing code points unassigned in Unicode version 4.0, and those denoting combining characters that could be parts of precomposed character added to later versions of Unicode be excluded from negative name caching. The necessary filtering could apply to all potential code points although clients might choose to simplify implementation by excluding strings containing code points beyond a certain point, e.g. (U+0FFFF).

When a client implements internationalized name handling, it needs to be able to detect when the apparent absence of a file within a directory is contradicted by the occurrence of a file with a distinct, but canonically equivalent, name. In order to efficiently find such names, when they exist, a client typically needs to implement a form of name hashing which always produces the same result for two canonically equivalent names. This can be done by making the contribution of any character to the name hash, equal to the contribution of the corresponding canonical decomposition string.

In the case of case-insensitive file systems, the uncertainty to be dealt with includes the version of Unicode implemented by the server as well as the details of the possible case-handling implemented by the server. In addition to the fact that different Unicode versions may have different canonical equivalence relationships, the server may implement different approaches to the handling of issues related to the handling of dotted and dotless i, in Turkish and Azeri. However, the question of whether the server's handling is case-preserving has no effect on client behavior, as is the question of whether the server implements a particular normalization form or implements form-insensitive file name matching. In light of the uncertainty created by the lack of knowledge of the details of the case-related equivalence relation together with the precise Unicode version used by the server to implement its canonical equivalence relation, the following possibilities, arranged in order of

increasing value (and difficulty of implementation) should be considered.

B1: The client can simply decline to implement optimizations based on negative name caching on case-insensitive file systems.

While this might have a negative effect on performance where significant benefits from negative name caching might be expected, it might be the best option for clients not heavily used to access case-insensitive file systems.

B2: Filtering similar to that discussed in item A2 could be implemented, although a higher limit is likely to be chosen (e.g. U+07FF) if significant use of non-Latin scripts is expected. Because of the uncertainty regarding the handling of case relationship among characters used for the variants of "I" used by Turkic languages, this filtering would have to exclude names containing LATIN CAPITAL LETTER I WITH DOT ABOVE and LATIN SMALL LETTER DOTLESS I together with precomposed characters derived from them.

In cases in which such filtering did not exclude the item from consideration, it would need to search for files with possibly equivalent names, including those equivalent by canonical equivalence, case-insensitive equivalence, or a combination of the two. This will typically require a form of name hashing which always produces the same hash for equivalent names, similar to that discussed in item A3 but including case-insensitive equivalence as well.

B3: A client might implement its own internationalized, case-insensitive file name handling paralleling that of the server. Because the case mappings are uncertain and the Unicode version used by the server file system is unknown, strings for which it is possible that the equivalent string might be different depending on the version of Unicode implemented by the server or the choice of case mappings would have to be identified and excluded from using negative name caching. This would require that strings containing code points unassigned in Unicode version 4.0, and those denoting combining characters that could be parts of precomposed characters added to later versions of Unicode be excluded from negative name caching. The necessary filtering could apply to all potential code points although clients might choose to simplify implementation by excluding strings containing code points beyond a certain point (e.g. U+00FFFF).

When a client implements internationalized name handling, it needs to be able to detect when the apparent absence of a file

within a directory is contradicted by the occurrence of a file with a distinct, but canonically equivalent name. In order to efficiently find such names, when they exist, a client typically needs to implements a form of name hashing which always produces the same result for two canonically equivalent names. This can be done by making the contribution of any character to the name hash, equal to contribution of the correspond canonical decomposition string.

11.3. Client Processing of File Names for Future NFSv4 Protocols

Because NFSv4 has an extension framework allowing the addition of new attributes in later minor versions or in extensions to extensible minor versions. Such new attributes are likely to be **OPTIONAL**. They could include a number of useful per-fs attributes to deal with the information gaps discussed in Section 11.2:

* The Unicode version used to define the canonical equivalence relation implemented by the server could be provided as an fs-scope attribute.

* For case-insensitive file systems, details regarding the actual case mapping used could be provided as an fs-scope attribute. These details would include the case mapping associated with LATIN LETTER I (i.e. whether the C-type or T-type case mappings or both are to be used). Similarly for characters having F-type case mappings, information needs to be provided about whether the F-type, mapping, the S-type mapping, or both, are to be used.

There is little prospect of such additional attributes being **REQUIRED**. Although the term "RECOMMENDED" has been used to describe NFSv4 attributes that are not **REQUIRED**, any such attributes are best considered **OPTIONAL** for the server to support with the client required to deal with the case in which the attribute is not supported.

When such attributes are defined and implemented, it would be possible for the client and server to implement compatible internationalization-related file name handling. However, as a practical matter, providing such compatibility would be considerably eased if there existed unencumbered open-source implementations of the algorithm and tables described in Appendix B. This would allow clients, servers, and server-based file systems, to easily adopt compatible approaches to these issues, each calling a common set of primitives, even though each might have a different execution environment and might be processing file names for different purposes.

In the case of a case-sensitive file system, the case-mapping attribute is not relevant. In dealing with the non-support of the Unicode version attribute, the client is in the same position as that of clients described in Section 11.2. In the case in which the Unicode version is supported, the client would be able to implement the same version of the canonical equivalence relation implemented by the server, thus avoiding the need for the sort of overbroad filtering mentioned in items A2 and A3 within Section 11.2

The case of case-insensitive file systems is more complicated, since there are two OPTIONAL attributes to deal with:

C1: When neither of these OPTIONAL attributes is supported, the client is in the same position as that of clients described in Section 11.2 in dealing with a case-insensitive file system.

C2: When the Unicode version is available but the details of case mapping are not, the client handling will be similar to that specified the options B1 through B3 defined in Section 11.2. However, in cases B2 and B3, it will be possible to reduce the scope of the character filtering applied, by enabling names containing characters defined after Unicode version 4.0 to be processed, as long as none of the case mapping options for those characters is at all problematic.

C3: When the details of case mapping are available but Unicode version is not, the client handling will be similar to that specified the options B1 through B3 defined in Section 11.2. However, in cases B2 and B3 However, in cases B2 and B3, it will be possible to reduce the scope of the character filtering by enabling names containing characters of uncertain case mapping to be processed as long as those character were defined in Unicode version 4.0.

C4: When both of these OPTIONAL attributes are supported, the client has the ability, at least theoretically, to reproduce the internationalization-related file name handling implemented by a server for a case-insensitive file system. However, when the client is unable to provide such an implementation, it is free to ignore the attribute and implement one of the options B1 through B3 defined in Section 11.2.

12. 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 12.1.

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 12.2.

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 apprear in the owner and group attributes and as ace_who values within ACL 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 opererations.

There is likely to be few or no implementations conformng 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 specifcation might have been different.

*When [RFC7530] was published, implementors 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 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 conists 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 and **MUST NOT** be done for strings that contain multi-byte Unicode characters.

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], RFC5661 [RFC5661])

12.1. 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 12.2.

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.

12.2. 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 5661 [RFC5661] 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 physical 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.

13. 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 14.

Where the string supplied by the client is not rejected with NFS4ERR_INVAL but contains characters that are not supported by the server as a value for that string (e.g., names containing slashes, or characters that do not fit into 16 bits when converted from UTF-8 to a Unicode codepoint), the server **MUST** return 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.

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

As stated previously, servers **MAY** accept, on all or on some subset of the physical file systems exported, component names that are not valid UTF-8 strings. A typical pattern is for a server to use UTF-8-unaware physical file systems that treat component names as uninterpreted strings of bytes, rather than having any awareness of the character set being used.

Such servers **SHOULD NOT** change the stored representation of component names from those received on the wire and **SHOULD** 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.

Nonetheless, when such a server uses a broader notion of string equivalence than what is recommended in the preceding paragraph, the following considerations apply:

* Outside of 7-bit ASCII, string processing that changes string contents is usually specific to a character set and hence is generally unsafe when the character set is unknown. This processing could change the file name in an unexpected fashion, rendering the file inaccessible to the application or client that created or renamed the file and to others expecting the original file name. Hence, such processing is best not performed, because

doing so is likely to result in incorrect string modification or aliasing.

*Unicode normalization is particularly dangerous, as such processing assumes that the string is UTF-8. When that assumption is false because a different character set was used to create the file name, normalization may corrupt the file name with respect to that character set, rendering the file inaccessible to the application that created it and others expecting the original file name. Hence, Unicode normalization **SHOULD NOT** be performed, because it might cause incorrect string modification or aliasing.

When the above recommendations are not followed, the resulting string modification and aliasing 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).

15. Future Minor Versions and Extensions

As stated above, all current NFSv4 minor versions allow use of non-UTF-8 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.

16. IANA Considerations

The current document does not require any actions by IANA.

17. 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 12.) are always encoded as UTF-8. String processing (e.g., Unicode normalization) raises security concerns for string comparison. See Sections 12 and 9 as well as the respective Sections 5.9 of RFC7530 [RFC7530] and RFC5661 [RFC5661] 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 used to identify the objects to which ACLs are applied (See the respective Sections 6 of RFC7530 [RFC7530] and RFC5661 [RFC5661]).

18. References

18.1. 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>.

18.2. Informative References

[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>.

[I-D.williams-filesystem-18n]
Williams, N., "Internationalization Considerations for Filesystems and Filesystem Protocols", Work in Progress,

Internet-Draft, draft-williams-filesystem-18n-00, 6 July 2020, <https://datatracker.ietf.org/doc/html/draft-williams-filesystem-18n-00>.

[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. 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 partiipants 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 7 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 RFC5661 [RFC5661] 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 9.

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 RFC5661, which specifies the internationalization for NFSv4.1, inherited by NFSv4.2.

*Later, as work on the predecessors of this document was underway, [I-D.williams-filesystem-18n] was submitted, making it necessary that some gaps 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 11.2 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 Section 11.3 discusses 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 B. 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 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, making this implementation guidance apropos despite any potential organizational barriers.

* Implementation of NFSv4 clients that need to provide matching internationalization-related handling for reason discussed in Section 11.

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 Section 10.2, 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.

* 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 Section 10.2, 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, need to provide information about the canonical decomposition of precomposed characters. See Appendix B.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.

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.5 for a discussion of how sets of combining characters can be compared).

B.1. Name Hashes

We discussed in Section 10.1 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 Section 10.1, 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 Section 10.2 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.

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.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. 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:

* 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 need 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 Section 10.2.

* 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 Section 10.2. 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.3. Outline of comparison

We are assuming that comparisons will be based on the hash values computed as described in Appendix B.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.4 and B.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.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.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.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 Section 10.2. These are best dealt with using the approach outlined in Section 10.1. 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 by 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 Section 10.2.

*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.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 immediate following 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 Section 10.2, 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 id designated the "source string" and successive character 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 fail 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.

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