This file documents version 2 of the svn protocol.

1. Syntactic structure

----------------------

The Subversion protocol is specified in terms of the following

syntactic elements, specified using ABNF [RFC 2234]:

item = word / number / string / list

word = ALPHA \*(ALPHA / DIGIT / "-") space

number = 1\*DIGIT space

string = 1\*DIGIT ":" \*OCTET space

; digits give the byte count of the \*OCTET portion

list = "(" space \*item ")" space

space = 1\*(SP / LF)

Here is an example item showing each of the syntactic elements:

( word 22 6:string ( sublist ) )

All items end with mandatory whitespace. (In the above example, a

newline provides the terminating whitespace for the outer list.) It

is possible to parse an item without knowing its type in advance.

Lists are not constrained to contain items of the same type. Lists

can be used for tuples, optional tuples, or arrays. A tuple is a list

expected to contain a fixed number of items, generally of differing

types. An optional tuple is a list containing either zero or a fixed

number of items (thus "optional" here does not refer to the list's

presence or absence, but to the presence or absence of its contents).

An array is a list containing zero or more items of the same type.

Words are used for enumerated protocol values, while strings are used

for text or binary data of interest to the Subversion client or

server. Words are case-sensitive.

For convenience, this specification will define prototypes for data

items using a syntax like:

example: ( literal ( data:string ... ) )

A simple word such as "literal", with no colon, denotes a literal

word. A choice of words may be given with "|" separating the choices.

"name:type" specifies a parameter with the given type.

A type is "word", "number", "string", "list", or the name of another

prototype. Parentheses denote a tuple, unless the parentheses contain

ellipses, in which case the parentheses denote an array containing

zero or more elements matching the prototype preceding the ellipses.

If a tuple has an optional part after the fixed part, a '?' marks

places where the tuple is allowed to end. The following tuple could

contain one, three, or four or more items:

example: ( fixed:string ? opt1:number opt2:string ? opt3:number )

Brackets denote an optional tuple; they are equivalent to parentheses

and a leading '?'. For example, this:

example: ( literal (? rev:number ) ( data:string ... ) )

can be written more compactly like this:

example: ( literal [ rev:number ] ( data:string ... ) )

For extensibility, implementations must treat a list as matching a

prototype's tuple even if the list contains extra elements. The extra

elements must be ignored.

In some cases, a prototype may need to match two different kinds of

data items. This case will be written using "|" to separate the

alternatives; for example:

example: ( first-kind rev:number )

| second-kind

The "command response" prototype is used in several contexts of this

specification to indicate the success or failure of an operation. It

is defined as follows:

command-response: ( success params:list )

| ( failure ( err:error ... ) )

error: ( apr-err:number message:string file:string line:number )

The interpretation of parameters in a successful command response is

context-dependent.

URLs and repository paths are represented as strings. They should be in

canonical form when sent over the protocol. However, as a matter of input

validation, an implementation should always canonicalize received paths if it

needs them in canonicalized form.

2. Connection establishment and protocol setup

----------------------------------------------

By default, the client connects to the server on port 3690.

Upon receiving a connection, the server sends a greeting, using a

command response whose parameters match the prototype:

greeting: ( minver:number maxver:number mechs:list ( cap:word ... ) )

minver and maxver give the minimum and maximum Subversion protocol

versions supported by the server. mechs is present for historical

reasons, and is ignored by the client. The cap values give a list of

server capabilities (see section 2.1).

If the client does not support a protocol version within the specified

range, it closes the connection. Otherwise, the client responds to

the greeting with an item matching the prototype:

response: ( version:number ( cap:word ... ) url:string

? ra-client:string ( ? client:string ) )

version gives the protocol version selected by the client. The cap

values give a list of client capabilities (see section 2.1). url

gives the URL the client is accessing. ra-client is a string

identifying the RA implementation, e.g. "SVN/1.6.0" or "SVNKit 1.1.4".

client is the string returned by svn\_ra\_callbacks2\_t.get\_client\_string;

that callback may not be implemented, so this is optional.

Upon receiving the client's response to the greeting, the server sends

an authentication request, which is a command response whose arguments

match the prototype:

auth-request: ( ( mech:word ... ) realm:string )

The mech values give a list of SASL mechanisms supported by the

server. The realm string is similar to an HTTP authentication realm

as defined in [RFC 2617]; it allows the server to indicate which of

several protection spaces the server wishes to authenticate in. If

the mechanism list is empty, then no authentication is required and no

further action takes place as part of the authentication challenge;

otherwise, the client responds with a tuple matching the prototype:

auth-response: ( mech:word [ token:string ] )

mech specifies the SASL mechanism and token, if present, gives the

"initial response" of the authentication exchange. The client may

specify an empty mechanism to decline authentication; otherwise, upon

receiving the client's auth-response, the server sends a series of

challenges, each a tuple matching the prototype:

challenge: ( step ( token:string ) )

| ( failure ( message:string ) )

| ( success [ token:string ] )

If the first word of the challenge is "step", then the token is

interpreted by the authentication mechanism, and the response token

transmitted to the server as a string. The server then proceeds with

another challenge. If the client wishes to abort the authentication

exchange, it may do so by closing the connection.

If the first word of the challenge is "success", the authentication is

successful. If a token is provided, it should be interpreted by the

authentication mechanism, but there is no response.

If the first word of the challenge is "failure", the authentication

exchange is unsuccessful. The client may then give up, or make

another auth-response and restart the authentication process.

RFC 2222 requires that a protocol profile define a service name for

the sake of the GSSAPI mechanism. The service name for this protocol

is "svn".

After a successful authentication exchange, the server sends a command

response whose parameters match the prototype:

repos-info: ( uuid:string repos-url:string ( cap:word ... ) )

uuid gives the universal unique identifier of the repository,

repos-url gives the URL of the repository's root directory, and the

cap values list the repository capabilities (that is, capabilities

that require both server and repository support before the server can

claim them as capabilities, e.g., SVN\_RA\_SVN\_CAP\_MERGEINFO).

The client can now begin sending commands from the main command set.

2.1 Capabilities

The following capabilities are currently defined (S indicates a server

capability and C indicates a client capability):

[CS] edit-pipeline Every released version of Subversion since 1.0

announces the edit-pipeline capability; starting

in Subversion 1.5, both client and server

\*require\* the other side to announce edit-pipeline.

[CS] svndiff1 If both the client and server support svndiff version

1, this will be used as the on-the-wire format for

svndiff instead of svndiff version 0.

[CS] absent-entries If the remote end announces support for this capability,

it will accept the absent-dir and absent-file editor

commands.

[S] commit-revprops If the server presents this capability, it supports the

rev-props parameter of the commit command.

See section 3.1.1.

[S] mergeinfo If the server presents this capability, it supports the

get-mergeinfo command. See section 3.1.1.

[S] depth If the server presents this capability, it understands

requested operational depth (see section 3.1.1) and

per-path ambient depth (see section 3.1.3).

[S] atomic-revprops If the server presents this capability, it

supports the change-rev-prop2 command.

See section 3.1.1.

3. Commands

-----------

Commands match the prototypes:

command: ( command-name:word params:list )

The interpretation of command parameters is different from command to

command.

Initially, the client initiates commands from the main command set,

and the server responds. Some commands in the main command set can

temporarily change the set of commands which may be issued, or change

the flow of control so that the server issues commands and the client

responds.

Here are some miscellaneous prototypes used by the command sets:

proplist: ( ( name:string value:string ) ... )

propdelta: ( ( name:string [ value:string ] ) ... )

node-kind: none|file|dir|unknown

bool: true|false

lockdesc: ( path:string token:string owner:string [ comment:string ]

created:string [ expires:string ] )

3.1. Command Sets

There are three command sets: the main command set, the editor command

set, and the report command set. Initially, the protocol begins in

the main command set with the client sending commands; some commands

can change the command set and possibly the direction of control.

3.1.1. Main Command Set

The main command set corresponds to the svn\_ra interfaces. After each

main command is issued by the client, the server sends an auth-request

as described in section 2. (If no new authentication is required, the

auth-request contains an empty mechanism list, and the server proceeds

immediately to sending the command response.) Some commands include a

second place for auth-request point as noted below.

reparent

params: ( url:string )

response: ( )

get-latest-rev

params: ( )

response: ( rev:number )

get-dated-rev

params: ( date:string )

response: ( rev:number )

change-rev-prop

params: ( rev:number name:string ? value:string )

response: ( )

If value is not specified, the rev-prop is removed.

(Originally the value was required; for minimum impact, it was

changed to be optional without creating an optional tuple for

that one parameter as we normally do.)

change-rev-prop2

params: ( rev:number name:string [ value:string ]

( dont-care:bool ? previous-value:string ) )

response: ( )

If value is not specified, the rev-prop is removed. If dont-care is false,

then the rev-prop is changed only if it is currently set as previous-value

indicates. (If dont-care is false and previous-value is unspecified, then

the revision property must be previously unset.) If dont-care is true,

then previous-value must not be specified.

rev-proplist

params: ( rev:number )

response: ( props:proplist )

rev-prop

params: ( rev:number name:string )

response: ( [ value:string ] )

commit

params: ( logmsg:string ? ( ( lock-path:string lock-token:string ) ... )

keep-locks:bool ? rev-props:proplist )

response: ( )

Upon receiving response, client switches to editor command set.

Upon successful completion of edit, server sends auth-request.

After auth exchange completes, server sends commit-info.

commit-info: ( new-rev:number date:string author:string

? ( post-commit-err:string ) )

get-file

params: ( path:string [ rev:number ] want-props:bool want-contents:bool )

response: ( [ checksum:string ] rev:number props:proplist )

If want-contents is specified, then after sending response, server

sends file contents as a series of strings, terminated by the empty

string, followed by a second empty command response to indicate

whether an error occurred during the sending of the file.

get-dir

params: ( path:string [ rev:number ] want-props:bool want-contents:bool

? ( field:dirent-field ... ) )

response: ( rev:number props:proplist ( entry:dirent ... ) )]

dirent: ( name:string kind:node-kind size:number has-props:bool

created-rev:number [ created-date:string ]

[ last-author:string ] )

dirent-field: kind | size | has-props | created-rev | time | last-author

| word

check-path

params: ( path:string [ rev:number ] )

response: ( kind:node-kind )

If path is non-existent, 'svn\_node\_none' kind is returned.

stat

params: ( path:string [ rev:number ] )

response: ( ? entry:dirent )

dirent: ( name:string kind:node-kind size:number has-props:bool

created-rev:number [ created-date:string ]

[ last-author:string ] )

New in svn 1.2. If path is non-existent, an empty response is returned.

get-mergeinfo

params: ( ( path:string ... ) [ rev:number ] inherit:word

descendents:bool)

response: ( ( ( path:string merge-info:string ) ... ) )

New in svn 1.5. If no paths are specified, an empty response is

returned. If rev is not specified, the youngest revision is used.

update

params: ( [ rev:number ] target:string recurse:bool

? depth:word send\_copyfrom\_param:bool )

Client switches to report command set.

Upon finish-report, server sends auth-request.

After auth exchange completes, server switches to editor command set.

After edit completes, server sends response.

response: ( )

switch

params: ( [ rev:number ] target:string recurse:bool url:string

? depth:word)

Client switches to report command set.

Upon finish-report, server sends auth-request.

After auth exchange completes, server switches to editor command set.

After edit completes, server sends response.

response: ( )

status

params: ( target:string recurse:bool ? [ rev:number ] ? depth:word )

Client switches to report command set.

Upon finish-report, server sends auth-request.

After auth exchange completes, server switches to editor command set.

After edit completes, server sends response.

response: ( )

diff

params: ( [ rev:number ] target:string recurse:bool ignore-ancestry:bool

url:string ? text-deltas:bool ? depth:word )

Client switches to report command set.

Upon finish-report, server sends auth-request.

After auth exchange completes, server switches to editor command set.

After edit completes, server sends response.

response: ( )

log

params: ( ( target-path:string ... ) [ start-rev:number ]

[ end-rev:number ] changed-paths:bool strict-node:bool

? limit:number

? include-merged-revisions:bool

all-revprops | revprops

? ( revprop:string ... ) )

Before sending response, server sends log entries, ending with "done".

If a client does not want to specify a limit, it should send 0 as the

limit parameter. rev-props excludes author, date, and log; they are

sent separately for backwards-compatibility.

log-entry: ( ( change:changed-path-entry ... ) rev:number

[ author:string ] [ date:string ] [ message:string ]

? has-children:bool invalid-revnum:bool

revprop-count:number rev-props:proplist )

| done

changed-path-entry: ( path:string A|D|R|M

? ( ? copy-path:string copy-rev:number )

? ( ? node-kind:string ? text-mods:bool prop-mods:bool ) )

response: ( )

get-locations

params: ( path:string peg-rev:number ( rev:number ... ) )

Before sending response, server sends location entries, ending with "done".

location-entry: ( rev:number abs-path:number ) | done

response: ( )

get-location-segments

params: ( path:string [ start-rev:number ] [ end-rev:number ] )

Before sending response, server sends location entries, ending with "done".

location-entry: ( range-start:number range-end:number [ abs-path:string ] ) | done

response: ( )

get-file-revs

params: ( path:string [ start-rev:number ] [ end-rev:number ]

? include-merged-revisions:bool )

Before sending response, server sends file-rev entries, ending with "done".

file-rev: ( path:string rev:number rev-props:proplist

file-props:propdelta ? merged-revision:bool )

| done

After each file-rev, the file delta is sent as one or more strings,

terminated by the empty string. If there is no delta, server just sends

the terminator.

response: ( )

lock

params: ( path:string [ comment:string ] steal-lock:bool

[ current-rev:number ] )

response: ( lock:lockdesc )

lock-many

params: ( [ comment:string ] steal-lock:bool ( ( path:string

[ current-rev:number ] ) ... ) )

Before sending response, server sends lock cmd status and descriptions,

ending with "done".

lock-info: ( success ( lock:lockdesc ) ) | ( failure ( err:error ) )

| done

response: ( )

unlock

params: ( path:string [ token:string ] break-lock:bool )

response: ( )

unlock-many

params: ( break-lock:bool ( ( path:string [ token:string ] ) ... ) )

Before sending response, server sends unlocked paths, ending with "done".

pre-response: ( success ( path:string ) ) | ( failure ( err:error ) )

| done

response: ( )

get-lock

params: ( path:string )

response: ( [ lock:lockdesc ] )

get-locks

params: ( path:string ? [ depth:word ] )

response ( ( lock:lockdesc ... ) )

replay

params: ( revision:number low-water-mark:number send-deltas:bool )

After auth exchange completes, server switches to editor command set.

After edit completes, server sends response.

response ( )

replay-range

params: ( start-rev:number end-rev:number low-water-mark:number

send-deltas:bool )

After auth exchange completes, server sends each revision

from start-rev to end-rev, alternating between sending 'revprops'

entries and sending the revision in the editor command set.

After all revisions are complete, server sends response.

revprops: ( revprops:word props:proplist )

(revprops here is the literal word "revprops".)

response ( )

get-deleted-rev

params: ( path:string peg-rev:number end-rev:number )

response: ( deleted-rev:number )

3.1.2. Editor Command Set

An edit operation produces only one response, at close-edit or

abort-edit time. However, the consumer may write an error response at

any time during the edit in order to terminate the edit operation

early; the driver must notice that input is waiting on the connection,

read the error, and send an abort-edit operation. After an error is

returned, the consumer must read and discard editing operations until

the abort-edit. In order to prevent TCP deadlock, the consumer must

use non-blocking I/O to send an early error response; if writing

blocks, the consumer must read and discard edit operations until

writing unblocks or it reads an abort-edit.

target-rev

params: ( rev:number )

open-root

params: ( [ rev:number ] root-token:string )

delete-entry

params: ( path:string rev:number dir-token:string )

add-dir

params: ( path:string parent-token:string child-token:string

[ copy-path:string copy-rev:number ] )

open-dir

params: ( path:string parent-token:string child-token:string rev:number )

change-dir-prop

params: ( dir-token:string name:string [ value:string ] )

close-dir

params: ( dir-token:string )

absent-dir

params: ( path:string parent-token:string )

add-file

params: ( path:string dir-token:string file-token:string

[ copy-path:string copy-rev:number ] )

open-file

params: ( path:string dir-token:string file-token:string rev:number )

apply-textdelta

params: ( file-token:string [ base-checksum:string ] )

textdelta-chunk

params: ( file-token:string chunk:string )

textdelta-end

params: ( file-token:string )

change-file-prop

params: ( file-token:string name:string [ value:string ] )

close-file

params: ( file-token:string [ text-checksum:string ] )

absent-file

params: ( path:string parent-token:string )

close-edit

params: ( )

response: ( )

abort-edit

params: ( )

response: ( )

finish-replay

params: ( )

Only delivered from server to client, at the end of a replay.

3.1.3. Report Command Set

To reduce round-trip delays, report commands do not return responses.

Any errors resulting from a report call will be returned to the client

by the command which invoked the report (following an abort-edit

call). Errors resulting from an abort-report call are ignored.

set-path:

params: ( path:string rev:number start-empty:bool

? [ lock-token:string ] ? depth:word )

delete-path:

params: ( path:string )

link-path:

params: ( path:string url:string rev:number start-empty:bool

? [ lock-token:string ] ? depth:word )

finish-report:

params: ( )

abort-report

params: ( )

4. Extensibility

----------------

This protocol may be extended in three ways, in decreasing order of

desirability:

\* Items may be added to any tuple. An old implementation will

ignore the extra items.

\* Named extensions may be expressed at connection initiation time

by the client or server.

\* The protocol version may be bumped. Clients and servers can then

choose to any range of protocol versions.

4.1. Extending existing commands

Extending an existing command is normally done by indicating that its

tuple is allowed to end where it currently ends, for backwards

compatibility, and then tacking on a new, possibly optional, item.

For example, diff was extended to include a new mandatory text-deltas

parameter like this:

/\* OLD \*/ diff:

params: ( [ rev:number ] target:string recurse:bool ignore-ancestry:bool

url:string )

/\* NEW \*/ diff:

params: ( [ rev:number ] target:string recurse:bool ignore-ancestry:bool

url:string ? text-deltas:bool )

The "?" says that the tuple is allowed to end here, because an old

client or server wouldn't know to send the new item.

For optional parameters, a slightly different approach must be used.

set-path was extended to include lock-tokens like this:

/\* OLD \*/ set-path:

params: ( path:string rev:number start-empty:bool )

/\* NEW \*/ set-path:

params: ( path:string rev:number start-empty:bool ? [ lock-token:string ] )

The new item appears in brackets because, even in the new protocol,

the lock-token is still optional. However, if there's no lock-token

to send, an empty tuple must still be transmitted so that future

extensions to this command remain possible.