User service daemon and client specification

Ian Jackson <ian@davenant.greenend.org.uk>

1.0.4

Abstract

This is a specification for a Unix system facility to allow one program to invoke another when only limited trust exists between them.

Copyright Notice

Contents

Chapter 1

Introduction

There is a daemon which invokes user service programs (henceforth 'services') in response to requests by callers of a companion client program (henceforth the 'client') and according to rules set forth in system-wide and user-specific configuration files. The companion client program is setuid root, and negotiates with the daemon through an `AF_UNIX` socket and associated objects in a system-wide private directory set aside for the purpose. The user who wishes the service to be performed and calls the client is called the 'calling user'; the process which calls the client is called the 'calling process'.

The daemon and the client are responsible for ensuring that information is safely carried across the security boundary between the two users, and that the processes on either side cannot interact with each other in any unexpected ways.

Chapter 2

Client program usage

```
userv options [--] service-user service-name [argument ...]
userv options -B|--builtin [--] builtin-service [info-argument ...]
```

service-user specifies which user is to provide the service. The user may be a login name or a numeric uid, or - to indicate that the service user is to be the same as the calling user.

The service name is interpreted by the userv[1] daemon on behalf of the service user. It will often be the name of a program.

2.1 Options

Single-letter options may be combined as is usual with Unix programs, and the value for such an option may appear in the same argument or in the next.

-B

--builtin Requests that a builtin service be provided. This is equivalent to using the `-override` option to specify a string consisting of `execute-builtin` followed by the *builtin-service* requested, and requesting a service user of - (indicating the calling user).

If the builtin service being requested requires a *service-argument* then this must be supplied to the client in the same argument as the *builtin-service*. See 'Directives for changing execution settings' on page 15 for details of the builtin services available, and 'Security-overriding options' on page 6 for details of the `-override` options.

The actual service name passed will be the *builtin-service*; note that this actual service name (as opposed to the override data) and the *info-argument*s supplied will be ignored by most builtin services; the override mechanism and `execute-builtin` will be used to ensure that the right builtin service is called with the right *service-argument*s.

[1] `userv` is short for 'user services', and is pronounced 'you-serve'.

`--file` *`fd[modifiers]=filename`* Requests that data be copied in and out of the service using pipes. For each file or descriptor this will be done by creating a pipe, one end of which is passed to the service program and the other end of which is passed to a copy of `cat` invoked by the client; the other file descriptor passed to `cat` will be one inherited by the client program from the caller or one opened by the client program on behalf of the caller.

The descriptor in the service program that should be connected must be specified as *fd*, either as a decimal number or as one of the strings `stdin`, `stdout` or `stderr`. The next argument is a filename which will be opened by the client with the privileges of the calling user.

modifiers is used to specify whether the file or descriptor is to be read from or written to. It consists of a series of words separated by commas. A comma may separate the *modifiers* from the *fd* and is required if *fd* is not numeric.

The modifier words are:

`read` `O_RDONLY`: Allow reading and not writing. May not be used with `write` or things that imply it.

`write` `O_WRONLY`: Allow writing and not reading. *Doesn't truncate or create* without `truncate` or `create`. `write` or things that imply it may not be used with `read`.

`overwrite` Equivalent to `write,create,truncate`.

`create`

`creat` `O_CREAT`: Creates the file if necessary. Implies `write`.

`exclusive`

`excl` `O_EXCL`: Fails if the file already exists. Implies `write` and `create`. May not be used with `truncate`.

`truncate`

`trunc` `O_TRUNC`: Truncate any existing file. Implies `write`. May not be used with `exclusive`.

`append` `O_APPEND`: All writes will append to the file. Implies `write` (but not `create`).

`sync` `O_SYNC`: Do writes synchronously. Implies `write`.

`wait`

`nowait`

`close` These modifiers control the behaviour of the client, with respect to the pipes carrying data to and from the service, when the service terminates. See below.

`fd` The *filename* is not a filename but a numeric file descriptor. One or both of `read` and `write` must be specified, and no other words are allowed. The *filename* may also be `stdin`, `stdout` or `stderr` for file descriptor 0, 1 or 2 respectively.

If no *modifiers* which imply `read` or `write` are used it is as if `write` had been specified, except that if the filedescriptor 0 of the service is being opened (either specified numerically or with `stdin`) it is as if `overwrite` had been specified (or `write` if only `fd` was specified).

The client will also use `O_NOCTTY` when opening files specified by the caller, to avoid changing its controlling terminal.

By default stdin, stdout and stderr of the service will be connected to the corresponding descriptors on the client. Diagnostics from the client and daemon will also appear on stderr.

If `wait` is specified, the client will wait for the pipe to be closed, and only exit after this has happened. This means that either the receiving end of the pipe connection was closed while data was still available at the sending end, or that the end of file was reached on the reading file descriptor. Errors encountered reading or writing in the client at this stage will be considered a system error and cause the client to exit with status 255, but will not cause disconnection at the service side since the service has already exited.

If `close` is specified the client will immediately close the pipe connection by killing the relevant copy of `cat`. If the service uses the descriptor it will get `SIGPIPE` (or `EPIPE`) for a writing descriptor or end of file for a reading one; the descriptor opened by or passed to the client will also be closed.

If `nowait` is specified then the client will not wait and the connection will remain open after the client terminates. Data may continue to be passed between the inheritors of the relevant descriptor on the service side and the corresponding file or descriptor on the client side until either side closes their descriptor. This should not usually be specified for stderr (or stdout if `--signals stdout` is used) since diagnostics from the service side may arrive after the client has exited and be confused with expected output.

The default is `wait` for writing file descriptors and `close` for reading ones.

-w*fd*=*action*

--fdwait*fd*=*action* Sets the action on termination of the service for the specified file descriptor; *action* must be `wait`, `nowait` or `close` as described above. The file descriptor must be specified as open when this option is encountered; this option is overridden by any later `-file` or `-fdwait` option - even by a `-file` which does not specify an action on termination (in this case the default will be used, as described above).

-D*name*=*value*

--defvar *name*=*value* Set a user-defined variable *name* to *value*. These user-defined variables are made available in the configuration language as the parameters `u-`*`name`* and are passed to the service in environment variables `USERV_U_`*`name`*. *name* may contain only alphanumerics and underscores, and must start with a letter. If several definitions are given for the same *name* then only the last is effective.

-t *seconds*

--timeout *seconds* Time out the service if it takes longer than *seconds* seconds (a positive integer, in decimal). Timeout will produce a diagnostic on stderr and an exit status of 255. If *seconds* is zero then no timeout will be implemented (this is the default).

--signals *method* Affects the handling of the exit status when the service terminates due to a signal. (The client will always finish by calling `_exit`, so that only numbers from 0 to 255 can be returned and not the full range of numbers and signal indications which can be returned by the `wait` family of system calls.)

The *method* may be one of the following:

status The client's exit status will be *status*. This will not be distinguishable from the service really having exited with code *status*. This method is the default, with a *status* of 254.

`number`

`number-nocore` The client's exit status will be the number of the signal which caused the termination of the service. If `number` is used rather than `number-nocore` then 128 will be added if the service dumped core. `number` is very like the exit code mangling done by the Bourne shell.

`highbit` The client's exit status will be the number of the signal with 128 added. If the service exits normally with an exit code of greater than 127 then 127 will be returned.

`stdout` The service's numeric wait status as two decimal numbers (high byte first) and a textual description of its meaning will be printed to the client's standard output. It will be preceded by a newline and followed by an extra newline, and the numbers are separated from each other and from the textual description by single spaces. The exit status of the client will be zero, unless a system error occurs in which case no exit status and description will be printed to stdout, and an error message will be printed to stderr as usual.

Problems such as client usage errors, the service not being found or permission being denied or failure of a system call are system errors. An error message describing the problem will be printed on the client's stderr, and the client's exit status will be 255. If the client dies due to a signal this should be treated as a serious system error.

`-H`

`--hidecwd` Prevents the calling process's current directory name from being passed to the service; the null string will be passed instead.

`-P`

`--sigpipe` If the service program is terminated due to a `SIGPIPE` the exit status of the client will be zero, even if it would have been something else according to the exit status method specified. This option has no effect on the code and description printed if the exit status method `stdout` is in use.

`-h`

`--help`

`--copyright` `-h` or `--help` prints the client's usage message; `--copyright` prints the copyright and lack of warranty notice.

2.2 Security-overriding options

There are also some options which are available for debugging and to allow the system administrator to override a user's policy. These options are available only if the client is called by root or if the calling user is the same as the service user.

--override *configuration-data*

--override-file *filename* Do not read the usual configuration files. Instead, the client sends *configuration-data* (followed by a newline) or the contents of *filename* (which is opened in the context of the client) to the daemon and the daemon uses that data instead. The *configuration-data* must all be in one argument. It will have a single newline appended so that a single directive can easily be given, but if more than one directive is required it will have to contain one or more real newlines.

--spoof-user *user* Pretend to the service that it is being called by *user* (which may be a username or a uid). This will also affect the group and supplementary groups supplied to the service; they will be the standard group and supplementary groups for *user*. The `--spoof-user` option will *not* affect which user is chosen if the service user is specified as just `-`; in this case the service user will be the real calling user.

Chapter 3

Execution environment of the service program

The daemon which is handling the service user side of things will read configuration files to decide what to do. If it decides to allow the service to be provided it will fork a subprocess to execute the service.

The service will have no controlling terminal, but it will be a process group leader.

If the client is killed or times out or a file or descriptor being read or written by the client process gets an error then the service will be disconnected from the client. The client will return an exit status of 255 and some the service's pipes may be closed at the other end. The service will become a child of `init`. The service may well not notice the disconnection, though writing to a pipe after this may produce a `SIGPIPE` and the facility exists to have a `SIGHUP` sent to the service on disconnection.

3.1 File descriptors

The service program's standard filedescriptors, and possibly other file descriptors, will be connected to pipes or to `/dev/null`. The `userv` client/daemon pair will arrange that data is copied between the files or file descriptors specified to to the client by the caller and these these pipes.

Pipes which may be written to will be closed if a write error occurs on the corresponding client-side file or descriptor, which may result in a `SIGPIPE` in the service program; pipes open for reading will get `EOF` if the client-side file descriptor gets `EOF` or an error.

If the service closes one of its reading file descriptors the writing end of the corresponding pipe will generate a `SIGPIPE` when attempts are made by the client/daemon pair to write to it. This will not be considered an error; rather, the relevant pipe will be discarded and the corresponding file or file descriptor held by the client will be closed.

Likewise, if one of the file descriptors held by the client for writing by the service is a pipe whose other end is closed by the caller then the client/daemon pair will see an error when trying to copy data provided by the service. This too will not be considered an error; rather, the pipe correspondong to that descriptor will be closed and any further writes will cause the service to get a `SIGPIPE`.

Note that not all write errors or broken pipes on file descriptors may be visible to the service, since buffered data may be discarded by the operating system and there will be a finite interval between the error happening and the service being disconnected from the client or the next write causing a `SIGPIPE`.

Read errors on file descriptors (and disconnection) will only be visible to the service and distinguishable from normal end of file if `disconnect-hup` is in effect.

Read and write errors (other than broken pipes, as described above) will always be visible to the caller; they are system errors, and will therefore cause the client to print an error message to stderr and return with an exit status of 255.

If the main service program process exits while it still has running children any file descriptors held by those children can remain open, depending on the use of `wait`, `nowait` or `close` for the relevant file descriptor in the client's arguments. By default writing filedescriptors remain open and the client will wait for them to be closed at the service end, and reading file descriptors are closed immediately. These leftover child processes will not get a any `SIGHUP` even if a read or write error occurs or the client disconnects before then.

3.2 Environment

The service will have some information in environment variables:

`USERV_USER` The login name of the calling user. If the `LOGNAME` variable is set (or, if that is unset, if the `USER` variable is set) in the environment passed to the client by the caller then the password entry for that login name will be looked up; if that password entry's uid is the same as that of the calling process then that login name will be used, otherwise (or if neither `LOGNAME` nor `USER` is set) the calling process's uid will be looked up to determine their login name (and if this lookup fails then the service will not be invoked).

`USERV_UID` The uid of the calling process.

`USERV_GID` The gid and supplementary group list of the calling process: first the group in gid and then those in the supplementary group list, in decimal, separated by spaces.

`USERV_GROUP` The group names of the calling process, listed in the same way as the ids are in `USERV_GID`. If no name can be found for any of the calling process's group(s) then the service will not be invoked.

`USERV_CWD` The client's current working directory name (this directory may not be accessible to the service). If it could not be determined or the `-hidecwd` flag was used then this variable will be set to an empty string (this is not considered an error).

`USERV_SERVICE` The service name requested by the caller.

`USERV_U_`*`name`* The value supplied to the client by the caller using -D*name*.

`HOME`, `PATH`, `SHELL`, `LOGNAME` and `USER` will be set appropriately (according to the details of the service user).

Chapter 4

Service-side configuration

Which services may be run by whom and under what conditions is controlled by configuration files.

The daemon will read these files in order. Certain directives in the files modify the daemon's execution settings for invoking the service, for example allowing certain file descriptors to be specified by the client or specifying which program to execute to provide the service.

The *last* instance of each such setting will take effect. The directives which specify which program to execute will not stop the configuration file from being read; they will be remembered and will only take effect if they are not overridden by a later directive.

The daemon will first read `/etc/userv/system.default`. Then, by default (this behaviour may be modified), it will read a per-user file `~/.userv/rc`, if it exists and the service user's shell is in `/etc/shells`. Finally it will read `/etc/userv/system.override`.

When it has read all of these files it will act according to the currently values of of the execution settings.

4.1 Configuration file syntax

The configuration file is a series of directives, usually one per line. The portion of a line following a hash character `#` is taken as a comment and ignored. Each directive consists of a series of tokens separated by linear whitespace (spaces and tabs); tokens may be words consisting of non-space characters, or, where a string is required, a string in double quotes. Double-quoted strings may contain the following backslash escapes:
`\n` newline
`\t` tab
`\r` carriage return
`\`*`OOO`* character whose octal code is *OOO*
`\x`*`XX`* character whose hex code is *XX*
`\`*`punctuation`* literal punctuation character (eg `\\`, `\"`)
`\`*`newline`* (ie, backslash at end of line) string continues on next line

Relative pathnames in directives are relative to the service program's current directory (usually the service user's home directory). Pathnames starting with the two characters `~/` are taken to be relative to the service user's home directory.

4.2 Configuration file directives

4.2.1 Immediate directives

The following directives take effect immediately:

`cd` ***`pathname`*** Change directory in the service program. `cd` is cumulative. It is an error if the directory cannot be changed to.

`cd` should not be used between `execute-from-directory` and the invocation of the service program, as the test for the availability of the service program would be done with the old current directory and the actual execution with the new (probably causing an error).

`eof` Stop reading the configuration file in question, as if end of file had been reached. Any control constructs (`if`, `catch-quit` or `errors-push`) which were started in that file will be considered finished. Parsing will continue in the file which caused the file containing the `eof` to be read.

`quit` Stop reading configuration files and act immediately on the current settings. The behaviour of `quit` is subject to the `catch-quit` control construct.

`include` ***`filename`***

`include-ifexist` ***`filename`*** Read the configuration file *filename*, and then return to this file and continue parsing it with the next directive. It is an error if the file cannot be opened and read, unless `include-ifexist` is used and the file does not exist, in which case the directive is silently ignored.

`include-lookup` ***`parameter directory`***

`include-lookup-all` ***`parameter directory`*** Read the configuration file in *directory* whose name is the value of *parameter* (see the description of `if`, 'Control structure directives' on the facing page). If *parameter* has several values they will be tried in order; with `include-lookup` this search will stop when one is found, but with `include-lookup-all` the search will continue and any files appropriate to other values will be read too.

If none of the parameter's values had a corresponding file then the file `:default` will be read, if it exists. If *parameter*'s list of values was empty then the file `:none` will be tried first and read if it exists, otherwise `:default` will be tried.

It is not an error for any of the files (including `:default`) not to exist, but it is an error if a file exists and cannot be read or if the directory cannot be accessed.

A translation will be applied to values before they are used to construct a filename, so that the lookup cannot access dotfiles or files in other directories: values starting with full stops will have a colon prepended (making `:.`), colons will be doubled, and each slash will be replaced with a colon followed by a hyphen `:-`. A parameter value which is the empty string will be replaced with `:empty` (note that this is different from a parameter not having any values).

`include-directory` ***`directory`*** Read configuration from all files in directory *directory* which are plain files whose names consist only of alphanumerics and hyphens and start with an alphanumeric. They will be read in lexical order. It is an error for the directory not to exist or for it or any of the files found not to be read successfully, or for anything with an appropriate name not to be a plain file or a symbolic link to a plain file.

`error` ***`text ...`*** Causes an error whose message includes the descriptive string *text*. *text* may consist of several tokens with intervening whitespace. The whitespace will be included in the message as found in the configuration file: all the characters until the end of the line will be included verbatim, unless they are part of a double-quoted string, in which case the usual meaning of the string (i.e., after backslash escape processing) will be used. Comments and linear whitespace at the end of the line (or just before the comment) will still be ignored.

`message` ***`text ...`*** Causes a message including the descriptive string *text* to be delivered as if it were an error message, but does not actually cause an error.

4.2.2 Directives with delayed effect

The following directives have no immediate effect, but are remembered and have an effect on later processing of the configuration files.

`user-rcfile` ***`filename`*** Specifies that the file *filename* should be read instead of the user's `~/.userv/rc`. This does *not* happen immediately; instead, the setting is remembered and used after the `system.default` configuration file has been read. This directive has no effect in a user's configuration file or in the `system.override` file, as the user's configuration file has already been found and read by then and will not be re-read.

`errors-to-stderr` Causes error messages to be delivered to the client's stderr.

`errors-to-file` *filename* Error messages will be written to *filename*, which will be opened in the context of and with the privileges of the service user.

`errors-to-syslog` [*facility* [*level*]] Error messages will be delivered using `syslog`. The default *facility* is `user`; the default *level* is `error`.

4.2.3 Control structure directives

The following directives are used to create control structures. If the end of the file is encountered before the end of any control structure which was started inside it then that control structure is considered finished. This is not an error.

fi Lines following `if` are interpreted only if the condition is true. Many conditions are properties of parameter values. Most parameters have a single string as a value; however, some may yield zero or several strings, in which case the condition is true if it is true of any of the strings individually. Parameters are described below.

The conditions are:

`glob` ***`parameter glob-pattern`*** **`...`** The value of the parameter whose name is given matches one of the glob patterns (anchored at both ends; backslashes can be used to escape metacharacters).

`range` ***`parameter min max`*** The value of the parameter is a nonnegative integer and lies within the range specified. *min* or *max* may be `$` to indicate no lower or upper limit, respectively.

`grep` ***`parameter filename`*** The *filename* refers to a file one of whose lines is the value of the parameter (leading or trailing whitespace on each line and empty lines in the file are ignored). It is an error for the file not to be opened and read.

`!` ***`condition`*** The *condition* is *not* true.

Conjunctions: `&` and `|`

```
( condition
& condition
& condition
...
)
```

is true if all the listed conditions are true; where | is used it is true if any of them is true. Newlines must be used to separate one condition from the next, as shown, and the parentheses are mandatory. These conjunctions do not do lazy evaluation.

The parameters are:

`service` The service name specified when the client was called.

`calling-user` Two strings: the login name of the calling user (determined as for `USERV_USER`, above) and the calling uid (represented in decimal).

`calling-group` Several strings: the primary and supplementary group names and gids (in decimal) of the calling process. All the group names come first, and then the gids. If the first supplementary group is the same as the primary group then it is elided.

`calling-user-shell` The calling user's shell, as listed in the password entry for the calling login name (as determined for `USERV_USER`, above).

`service-user` Two strings: the name of the service user (as specified to the client) and their uid (represented in decimal).

`service-group` Several strings: the primary and supplementary group names and gids (in decimal) of the service user.

`service-user-shell` The service user's shell, as listed in their password entry.

`u-`*`name`* The value of the user-defined variable *name* passed by the caller using the `-defvar` command-line option to the client. If the variable was not defined then this parameter is an empty list of strings; in this case any condition which tests it will be false, and `include-lookup` on it will read the `:none` file, or `:default` if `:none` is not found.

`errors-push` *filename*

`srorre` Stacks the error handling behaviour currently in effect. Any changes to error handling will take effect only between `errors-push` and `srorre`.

`catch-quit`

`hctac` Any use of `quit` inside `catch-quit` will merely cause the parsing to continue at `hctac` instead. Any control constructs started since the `catch-quit` will be considered finished if a `quit` is found.

If an error occurs inside `catch-quit` the execution settings will be reset (as if by the `reset` directive) and parsing will likewise continue at `hctac`.

If a lexical or syntax error is detected in the same configuration file as the `catch-quit`, while looking for the `hctac` after an error or `quit`, that new error will not be caught.

4.2.4 Directives for changing execution settings

The following directives modify the execution settings; the server will remember the fact that the directive was encountered and act on it only after all the configuration has been parsed. The *last* directive which modifies any particular setting will take effect.

`reject` Reject the request. `execute`, `execute-from-directory` and `execute-from-path` will change this setting.

`execute` *`program`* `[`*`argument`* `...]` Execute the program *program*, with the arguments as specified, followed by any arguments given to the client if `no-suppress-args` is in effect. It is an error for the execution to fail when it is attempted (after all the configuration has been parsed). If *program* does not contain a slash it will be searched for on the service user's path.

`execute-from-directory` *`pathname`* `[`*`argument`* `...]` Take all the characters after the last slash of the service name specified when the client was called, and execute that program in the directory named by *pathname* as if it had been specified for *execute*. The part of the service name used may contain only alphanumerics and hyphens and must start with an alphanumeric (and it must be non-empty), otherwise it is an error.

This directive is ignored if the relevant program does not exist in the directory specified; in this case the program to execute is left at its previous setting (or unset, if it was not set before).

It is an error for the test for the existence of the program to fail other than with a 'no such file or directory' indication. It is also an error for the execution to fail if and when it is attempted (after all the configuration has been parsed).

`execute-from-path` *service* is interpreted as a program on the default `PATH` (or as a pathname of an executable, if it contains a `/`). This directive is *very dangerous*, and is only provided to make the `-override` options effective. It should not normally be used. It is an error for the execution to fail when it is attempted (after all the configuration has been parsed).

execute-builtin *service-name service-arguments* Executes the builtin service *service-name*. These builtin services display information about the server and/or the request, and ignore any arguments passed from the service side except possibly to print them as part of their output. They write their results to their standard output (i.e., wherever file descriptor 1 is directed). The builtin services are:

- **execute** Displays the execution settings, defined variables, arguments, etc. with which the builtin service was invoked.
- **environment** Displays the environment variable settings with which the builtin service was invoked.
- **parameter *parameter*** Displays the values of the service configuration language parameter specified.
- **version** Displays the version string and compilation details of the uservd server program.
- **reset** Displays the default reset configuration (evaluated when `reset` is found in a configuration file, or when an error is caught by `catch-quit`).
- **toplevel** Displays the top-level default configuration (the configuration data, evaluated by the server, which calls all the other configuration files).
- **override** Displays the top-level override configuration (the configuration data, evaluated by the server, which causes all the other configuration data to be parsed).
- **help** Displays a list of the understood builtin service names and arguments.

In the future other builtin services may be defined which do more than just print information.

set-environment

no-set-environment Runs `/etc/environment` to set the service user's environment. This adds the overhead of invoking a shell, but doesn't cause any shell (de)mangling of the service's arguments. This is achieved by invoking

```
.../program arg arg arg ...
```

as

```
/bin/sh -c '. /etc/environment; exec "$@"' - .../program arg arg arg ...
```

`no-set-environment` cancels the effect of `set-environment`.

no-suppress-args

suppress-args Include any arguments given to the client as arguments to the program invoked as a result of an `execute`, `execute-from-directory` or `execute-from-path` directive. `suppress-args` undoes the effect of `no-suppress-args`.

require-fd *fd-range* read|write Insist that the filedescriptor(s) be opened for reading resp. writing. It is an error if any descriptor marked as required when the service is about to be invoked (after the configuration has been parsed) was not specified when the client was invoked. Each file descriptor has a separate setting, and the last one of `require-fd`, `allow-fd`, `ignore-fd`, `null-fd` or `reject-fd` which affected a particular file descriptor will take effect.

fd-range may be a single number, two numbers separated by a hyphen, or one number followed by a hyphen (indicating all descriptors from that number onwards). It may also be one of the words `stdin`, `stdout` or `stderr`. Open-ended file descriptor rangers are allowed only with `reject-fd` and `ignore-fd`, as otherwise the service program would find itself with a very large number of file descriptors open.

When the configuration has been parsed, and before the service is about to be executed, stderr (fd 2) must be required or allowed (`require-fd` or `allow-fd`) for writing; this is so that the error message printed by the server's child process if it cannot `exec` the service program is not lost.

`allow-fd` ***`fd-range`*** **`[read|write]`** Allow the descriptor(s) to be opened for reading resp. writing, or either if neither `read` nor `write` is specified. If a particular descriptor not specified by the client then it will be open onto `/dev/null` (for reading, writing, or both, depending on whether `read`, `write` or neither was specified).

`null-fd` ***`fd-range`*** **`[read|write]`** Specify that the descriptor(s) be opened onto `/dev/null` for reading resp. writing, or both if neither `read` nor `write` is specified. Any specification of these file descriptors by the client will be silently ignored; the client will see its ends of the descriptors being closed immediately.

`reject-fd` ***`fd-range`*** Do not allow the descriptor(s) to be specified by the client. It is an error if any descriptor(s) marked for rejection are specified when the service is about to be invoked (after the configuration has been parsed).

`ignore-fd` ***`fd-range`*** Silently ignore any specification by the client of those descriptor(s). The pipes corresponding to these descriptors will be closed just before the service is invoked.

`disconnect-hup`

`no-disconnect-hup` Causes the service's process group to get a `SIGHUP` if the client disconnects before the main service process terminates. `no-disconnect-hup` cancels `disconnect-hup`.

If one of the reading descriptors specified when the client is called gets a read error, or if the service is disconnected for some other reason, then the `SIGHUP` will be delivered *before* the writing end(s) of the service's reading pipe(s) are closed, so that the client can distinguish disconnection from reading EOF on a pipe.

`reset` Resets the execution settings to the default. This is equivalent to:

```
cd ~/
```

```
reject
no-set-environment
suppress-args
allow-fd 0 read
allow-fd 1-2 write
reject-fd 3-
disconnect-hup
```

If no `execute`, `execute-from-path`, `execute-from-directory` or `builtin` is interpreted before all the files are read then the request is rejected.

4.3 Errors in the configuration file

If a syntax error or other problem occurs when processing a configuration file then a diagnostic will be issued, to wherever the error messages are currently being sent (see the `errors-` family of directives, above).

The error will cause processing of the configuration files to cease at that point, unless the error was inside a `catch-quit` construct. In this case the settings controlling the program's execution will be reset to the defaults as if a `reset` directive had been issued, and parsing continues after `hctac`.

4.4 Defaults

The default configuration processing is as if the daemon were parsing an overall configuration file whose contents were as follows:

```
reset
user-rcfile ~/.userv/rc
errors-to-stderr
include /etc/userv/system.default
if grep service-user-shell /etc/shells
   errors-push
     catch-quit
       include-ifexist file specified by most recent user-rcfile directive
     hctac
   srorre
fi
include /etc/userv/system.override
quit
```

If one of the `-override` options to the client is used then it will instead be as if the daemon were parsing an overall configuration as follows:

```
reset
errors-to-stderr
include file containing configuration data sent by client
quit
```

Chapter 5

Information passed through the client/daemon combination

The information described below is the only information which passes between the caller and the service.

- The service name supplied by the caller is available in the configuration language for deciding whether and which service program to invoke, in the `service` parameter, and is used by the `execute-from-directory` and `execute-from-path` configuration directives. It is usually used to select which service program to invoke. It is also passed to the service program in the `USERV_SERVICE` environment variable.

- File descriptors specified by the client and allowed according to the configuration language will be connected. Each file descriptor is opened for reading or writing. Communication is via pipes, one end of each pipe being open on the appropriate file descriptor in the service program (when it is invoked) and the other end being held by the client process, which will read and write files it opens on behalf of its caller or file descriptors it is passed by its caller.

  Data may be passed into the service through reading pipes and out of it through writing pipes. These pipes can remain open only until the service and client have terminated, or can be made to stay open after the client has terminated and (if the service program forks) the main service process has exited; the behaviour is controlled by options passed to the client by its caller.

  The caller can arrange that a writing pipe be connected to a pipe or similar object and cause attempts to write to that descriptor by the service to generate a `SIGPIPE` (or `EPIPE` if `SIGPIPE` is caught or ignored) in the service.

  Likewise, the service can close filedescriptors specified for reading, which will cause the corresponding filedescriptors passed by the caller to be closed, so that if these are pipes processes which write to them will receive `SIGPIPE` or `EPIPE`.

- If `no-suppress-args` is set then arguments passed to the client by its caller will be passed on, verbatim, to the service.

- Fatal signals and system call failures experienced by the client will result in the disconnection of the service from the client and possibly some of the communication file descriptors described above; if `disconnect-hup` is set then the service will also be sent a `SIGHUP`.
- The value of the `LOGNAME` (or `USER`) environment variable as passed to the client will be used as the login name of the calling user if the uid of the calling process matches the uid corresponding to that login name. Otherwise the calling uid's password entry will be used to determine the calling user's login name.

  This login name and the calling uid are available in the configuration language in the `calling-user` parameter and are passed to the service program in environment variables `USERV_USER` and `USERV_UID`.

  The shell corresponding to that login name (according to the password entry) is available as in the configuration language's `calling-user-shell` parameter.

  If no relevant password entry can be found then no service will be invoked.
- The numeric values and textual names for calling gid and supplementary group list are available in the configuration language in the `calling-group` parameter and are passed to the service in environment variables.

  If no name can be found for a numeric group to which the calling process belongs then no service will be invoked.
- The name of the current working directory in which the client was invoked is passed, if available and not hidden using `-hidecwd`, to the service program in the `USERV_CWD` variable. This grants no special access to that directory unless it is a subdirectory of a directory which is executable (searchable) but not readable by the service user.
- Settings specified by the caller using the `--defvar` *`name`*`=`*`value`* option to the client are available in the configuration language as the corresponding `u-`*`name`* parameters and are passed to the service program in environment variables `USERV_U_`*`name`*.
- If the calling user is root or the same as the service user then options may be given to the client which bypass the usual security features; in this case other information may pass between the caller and the service.

Chapter 6

Applications and notes on use

6.1 Examples

The companion package, `userv-utils`, contains a selection of example services, some of which are useful tools in their own right. See the `README` in its top-level directory for details.

6.2 Standard services and directory management

In later versions of this specification standard service names and interfaces for common services such as mail delivery and WWW CGI scripts may be specified.

`userv`-using applications and system services which hide `userv` behind wrapper scripts may need to store information in the user's filespace to preserve the correct placement of the security perimiters. Such applications should usually do so in a directory (created by them) `~/.userv/`*`service`*, where *service* is the service name or application in question.

If desired, a dot-directory inside `~/.userv` may be used to avoid the user becoming confused by finding parts of a semi-privileged application's internal state in their filespace, and/or discourage them from fiddling with and thus corrupting it.

However, `userv` applications should of course not rely for their global integrity and security on the integrity of the data on the user's side of the security boundary.

6.3 Reducing the number of absolutely privileged subsystems

Currently most Unix systems have many components which need to run as root, even though most of their activity does not strictly require it. This gives rise to a large and complex body of code which must be trusted with the security of the system.

If they were to use `userv`, many of these subsystems would no longer need any unusual privilege.

cron and at, lpr and the system's mail transfer agent (sendmail, smail, exim or the like) all fall into this category, though userv-based versions of these programs are not currently available.

6.4 Do not give away excessive privilege to **userv**-using facilities

There is a danger that people reimplementing the facilities I mention above using userv will discard much of the security benefit by using a naive implementation technique. This will become clearer with an example:

Consider the lpr program. In current systems this needs to have an absolutely privileged component in order to support delayed printing without copying: when the user queues a file to be printed the filename is stored in the print queue, rather than a copy of it, and the printer daemon accesses the file directly when it is ready to print the job. In order that the user can print files which are not world-readable the daemon is given root privilege so that it can open the file in the context of the user, rather than its own.

A simple-minded approach to converting this scheme to use userv might involve giving the printer daemon (the lp user) the ability to read the file by allowing them to run cat (or a special-purpose file-reading program) as any user. The lpr program would use a userv service to store the filename in the printer daemon's queues, and the daemon would read the file later when it felt like it.

However, this would allow the printer daemon to read any file on the system, whether or not someone had asked for it to be printed. Since many files will contain passwords and other security-critical information this is nearly as bad as giving the daemon root access in the first place. Any security holes in the print server which allow a user to execute commands as the lp user will give the user the ability to read any file on the system.

Instead, it is necessary to keep a record of which files the daemon has been asked to print *outside* the control of the print daemon. This record could be kept by a new root-privileged component, but this is not necessary: the record of which files a user has asked to be printed can be kept under the control of the user in question. The submission program lpr will make a record in an area under the user's control before communicating with the print server, and the print server would be given the ability to run a special file-reading program which would only allow files to be read which were listed in the user's file of things they'd asked to print.

Now security holes in most of the printing system do not critically affect the security of the entire system: they only allow the attacker to read and interfere with print jobs. Bugs in the programs run by the print server to read users' files (and to remove entries from the list of files when it has done with them) will still be serious, but this program can be quite simple.

Similar considerations apply to many userv-based versions of facilities which currently run as root.

It is debatable whether the user-controlled state should be kept in the user's filespace (in dotfiles, say) or kept in a separate area set aside for the purpose; however, using the user's home directory (and possibly creating a separate subdirectory of it as a dotfile to contain subsystem state) has fewer implications for the rest of the system and makes it entirely clear where the security boundaries lie.

6.5 `userv` can often replace `sudo`, but not `really`

`userv` is not intended as a general-purpose system administration tool with which system administrators can execute arbitrary programs like text editors as root (or other system users) when they need to. It is unsuitable for this purpose precisely because it enforces a strong separation between the calling and the called program, which is undesirable in this context.

However, its use when restricted to running particular programs in particular ways is very similar to many common uses of `sudo`[1]. `userv` is generally much better than restricted `sudo`, because it protects the called program much more strongly from bad environmental conditions set up by the caller. Most programs that one might want to run via restricted `sudo`, have not been designed to run in a partially hostile environment. `userv` allows these programs to be run in a safer environment and should be used instead.

6.6 Error handling and input streams (eg stdin)

When the service program is reading from a file descriptor connected to the calling side, the fd that the service program refers to a pipe set up by `userv` and not to the same object as was presented by the caller.

Therefore if there is some kind of error it is possible for the service-side fd to give premature end of file. If it is important to tell whether all of the intended data has been received by the service program, the datastream must contain an explicit end-of-file indication of some kind.

For example, consider a `userv` service for submitting a mail message, where message is supplied on the service's stdin. However, if the calling process is interrupted before it has written all of the message, the service program will get EOF on the message data. In a naive arrangement this would cause a half-complete message to be sent. To prevent this, it is necessary to adopt some kind of explicit end indication; for example, the end of the message could be signalled by a dot on a line by itself, and dots doubled, as in SMTP. Then the service program would know when the entire message had been received, and could avoid queueing incomplete messages.

6.7 Don't give access to general-purpose utilities

Do not specify general purpose programs like `mv` or `cat` in `execute-` directives without careful thought about their arguments, and certainly not if `no-suppress-args` is specified. If you do so it will give the caller much more privilige than you probably intend.

[1] `sudo` is a program which allows users to execute certain programs as root, according to configuration files specified by the system administrator.

It is a shame that I have to say this here, but inexperienced administrators have made similar mistakes with programs like `sudo`.