X Consortium J. Brown
INTERNET-DRAFT Quarterdeck Office Systems
August 1993

A Flexible Remote Execution Protocol Based on **rsh**

Status of this Memo

This document is being distributed to members of the Internet community in order to solicit their reactions to the proposals contained in it. This memo does not specify an Internet Standard. Distribution of this memo is unlimited.

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months. Internet-Drafts may be updated, replaced, or obsoleted by other documents at any time. It is not appropriate to use Internet-Drafts as reference material or to cite them other than as a "working draft" or "work in progress."

To learn the current status of any Internet-Draft, please check the 1id-abstracts.txt listing contained in the Internet-Drafts Shadow Directories on ds.internic.net, nic.nordu.net, ftp.nisc.sri.com, or munnari.oz.au.

This document is filed under "(filename here)".

Background:

One of the X Window System's main strengths is the ability to display and control graphical applications remotely. It does not, however, address the issue of how to start these applications - the application initiates the connection to the server sitting in front of the user, not the other way 'round. Some other, non-X, mechanism must be used to start the application.

Current methods and their problems:

1) Walk over to the other machine, log in, and run something with its display redirected to your workstation. Not very appealing, but simple. No security problems associated with whether or not you should be able to run something on that machine.

2) Telnet to the other machine, log in, and run something with its display redirected to your workstation. Relatively simple, but doesn't allow for starting remote apps using a menu item - requires either human intervention or an autologin mechanism with its associated troubles. Security implication is that the password is passed "raw" over the telnet connection and is susceptible to snooping.

3) Use one of the non-standard Berkeley UNIX† "remote command" facilities, or their Kerberized equivalents - rsh, rexec, krsh. These can provide for the no-human-involved startup desired for menus, but have security implications which have been adequately documented elsewhere and

† UNIX is a trademark of Bell Laboratories.

require tuning to achieve ‘‘desirable’’ operation, or indeed operation at all.

4) Other proprietary schemes.

Well, (1) is obviously not very pleasant. (2) isn’t much better, because you can’t run things off menu items. Neither (1) nor (2) is ‘‘friendly’’ to an automatic session-restart system. (4), being proprietary, isn’t of much interest in a wide context. (Note that in any of these cases once you’ve started an app, an xterm say, you can ask it to start others. This is OK for some people, but not very acceptable if you want things picked off menus and especially not if you want apps on various machines picked off the same menu. It also isn’t friendly to automatic session-restart schemes.)

(3) is what this memo discusses.

Problems:

- Systems and shells vary in what exactly constitutes a ‘‘command’’ and what the syntax is.
- The apps might not be in the ‘‘system default search path’’.
- The apps might require additional setup before they can run - environment variables, etc. (LD_LIBRARY_PATH with OpenWindows 2 on Suns)
- Resources (TCP connections, processes, etc) get held open on both ends to support the (mostly-idle) rsh connection.
- No standard way to pass environment variables - DISPLAY, SESSION_MANAGER, etc.

These issues, both in their complexity and in their system-specificity, cause continual trouble for less technically aware users and are a nuisance even for technical users. In addition, more sophisticated uses of remote execution mechanisms (session restart for instance) may require more control over the environment than is available through the normal rsh mechanism.

The result is that a great deal of system-dependent manual tuning is required to achieve a pleasant result, where the DISPLAY and SESSION_MANAGER values (and others in the future) are passed transparently, where the app actually gets started(!), where unnecessary resources are not held open, etc.

The obvious answer is to define a ‘‘better’’ remote execution mechanism, but that has a host of problems - while rsh and friends have their security problems, at least they are reasonably well understood and the ‘‘trusted’’ executables have been extensively tested. A new mechanism would have to be analysed and tested before it could be trusted.

Better to make effective use of current services, and take advantage of such services as may become available in the future. In the future some better remote execution protocol may become available. Such a protocol may address some of the issues this paper is intended to address like passing auxiliary information, but will probably not address issues like setting up the environment required to run an X app. Fine, adopt it, and adopt conventions like those discussed here to allow its convenient use for X applications.

In this spirit this paper does not mandate ‘‘thou shalt start X apps using rsh’’, but rather ‘‘If you support X, you should arrange that the following rsh request will start an X app properly’’.

Such a solution would be to layer a more flexible protocol on top of rsh et al, using rsh’s standard in/out mechanism to pass initial setup information to a ‘‘helper’’ program on the other end. Since rsh (or whatever) has already handled the security aspects of the request, the helper need have no particular special privileges and hence adds no new security considerations.

‘‘Why rsh?’’ While this paper refers to ‘‘rsh’’ all over the place, few if any of the concepts are peculiar to rsh. Any form of remote execution protocol that handles the security aspects of remote execution and provides a bidirectional data stream to the resulting program can be used. Rsh, rexec, and krsh are obvious examples, but there is no reason why telnet could not be used, when combined with a scripting mechanism to do autologin.

‘‘But I don’t use X, why do I want this?’’ Again, while this paper refers to X all over the place and the impetus for creation of this protocol is starting and restarting X applications, the mechanisms defined are as independent of X as possible and are applicable to non-X start/restart problems.

‘‘I don’t use POSIX. What do I do?’’ The initial target systems are generally POSIX-based or POSIX-like, and so there are some POSIX-specific features and lots of POSIX-specific examples, but the protocol is designed to be operating-system independent. Such OS independence is, in fact, one of the primary goals - to provide an OS-independent remote execution mechanism.

The Remote Start Protocol ‘‘rstart’’

Goals:

- A requester should be able to have a program started in any of a number of predefined ‘‘contexts’’. (For instance, on a dual-universe Berkeley UNIX / System V UNIX system those might be two such contexts. On a system with multiple versions of the X Window System installed each would be available as a predefined context. A VAX might support VMS and Eunice contexts.)
- A requester should be able to override (within security bounds usual to the system) any aspect of the environment.
- Neither the requesting program nor the ‘‘helper’’ program on the host end should need to have any special privileges.
- Any parameter of the environment that can be controlled should be controllable through this mechanism. In particular, on POSIX systems environment variables, open files(?), umask, current directory, etc should all be controllable.
- The full richness of the host’s command argument mechanism should be available. In particular, on POSIX this means that arguments may contain any character and may be of any length.
- Notwithstanding all of the control afforded the requester, none of it should be required - the requester should be able to provide simple ‘‘command lines’’ which will be executed in the desired environment much as if they were typed to a conventional command processor.
- A ‘‘Generic Command’’ mechanism is provided, where standardized commands result in an appropriate response, independent of the host system. (This might be considered to be similar to the FTP model, where the LIST command will produce a directory listing no matter what the underlying system.)

Requesting System Requirements:

The requesting system MUST support an ‘‘rsh’’ client or a suitable replacement. Support of the rsh standard error and control connection is desirable but not essential.

Host System Requirements:

The host system MUST support an ‘‘rsh’’ server or a suitable replacement. Support of the rsh standard error and control connection is desirable but not essential. Invocation of the ‘‘rstartd’’ program in the default rsh environment MUST ‘‘work’’. On POSIX systems this might require that rstartd be installed in one of the directories in the default $PATH and that rstartd not need any non-default shared library setup, etc. On other systems it might require that the rsh server specially recognize the string ‘‘rstartd’’ and take appropriate action.

The protocol itself:

The requesting system makes a remote execution request using rsh (or other suitable protocol) to execute the program ‘‘rstartd’’. The host system responds with any amount of data (to accomodate systems that want to natter before starting a program), and then sends a line consisting of (exactly):

    rstartd:<space>Ready:<space><welcome/version message>

Failure to receive this line before the connection closes indicates that the host system does not support Extended rsh. A timeout is probably appropriate in addition.

The requester then sends a series of lines of ASCII specifying the program to be run and the environment in which it is to be run. The request is terminated by a blank line.

Syntax: rstart data is passed as a series of lines of ASCII, with spaces delimiting words in each line, terminated by a blank line. A ‘‘line’’ is terminated by an ASCII LF. CRs and NULs MUST be ignored, to allow for a system transmitting in Internet NVT ASCII. (Yes, rsh et al are POSIX-style tools and won’t have CRs, but if there’s ever an Internet standard remote exec mechanism it will almost certainly require NVT ASCII, so it seems wise to provide for that possibility from the start.) It is explicitly allowed for a system to discard characters other than LF outside the range decimal 32-126; characters outside that range MUST NOT be used. Words may contain spaces and non-printable characters by representing them as octal escape sequences consisting of a backslash followed by three octal digits. Note that backslashes themselves MUST be passed using this mechanism. Thus the initial parsing sequence consists of:

1) Receive data until the first blank line.
2) Break data into lines at LFs, discarding CRs and NULs.
3) Break lines into words at spaces.
4) Translate \nnn sequences in words into the appropriate characters.
5) Process each line as appropriate.
6) Pass (or more likely, allow to be passed) the connection and any data after the blank line to the program. (??? Hmm. stdio buffering considerations. Byte count instead of blank line?)

The first word of each line is a keyword specifying the interpretation of the line. Keywords SHOULD be transmitted as shown in this document, but the receiver MUST accept them in any case, even mIxEd.

Unless otherwise specified, only one instance of any given keyword is permitted in a given request. CONTEXT MUST be specified first, and after that keywords may be given in any order.

After receiving the blank line, the host responds with any number of lines of output of the following forms, terminated by a blank line.

rstartd: Ready: <message>

(This isn’t one of the ‘‘response’’ lines, but it’s included here for completeness.) This is rstartd’s ‘‘hello’’ line, and confirms that the host does indeed support rstartd.

rstartd: Failure: <message>

An unrecoverable error has occurred which indicates that either the requester or the host is fatally misconfigured. This might occur if, for instance, a request is malformed or a required configuration file is not present.

rstartd: Error: <message>

An unrecoverable error has occurred which indicates that the request is in error. The most common such error would be that the requested program is unavailable.

rstartd: Warning: <message>

A recoverable error has occurred. The program will be executed but may not behave as desired.

rstartd: Success: <message>

No errors were detected. Unfortunately this does not mean that no errors will occur, because there are many classes of errors that cannot be detected until rstartd actually attempts to pass control to the program.

rstartd: Debug: <message>

A debug message. Programs (and most humans!) should ignore these.

<anything else>

Indicates that something else in the system just HAD to say something; should be treated as a Warning.

Keywords

CONTEXT *name*

Initializes defaults suitable for the specified context. See the section on contexts for more information.

CONTEXT must be present, and must be the first line of the request.

CMD *command args args args*

Executes the specified command with the specified arg in the same general way as it would have been executed if typed to a the user's command interpreter. (If the user's primary interface is not a command language, a system default command language should be used.) It is expected that the command will have been provided by a user, who will expect ''normal'' command language handling of it. (For POSIX people, consider this as roughly equivalent to system().)

Note: No particular parsing or interpretation is guaranteed. The interpretation should be unsurprising to an ordinary user of the host system. This mechanism is therefore unsuitable for completely automated use; EXEC and GENERIC-CMD are provided for that purpose.

Exactly one of CMD, EXEC, or GENERIC-CMD must be present.

EXEC *progname namearg arg arg arg ...*

Executes a program using a low-level execution mechanism which provides minimal interpretation; in particular a command processor should not enter the picture and no quoting other than that required by this protocol should be required. It is expected that this interface will be used by programs requesting restart; presumably they know exactly what their desired arguments are and a command processor will only confuse the issue. (For POSIX people, consider this to be like execlp().)

Progname is the name of the executable. This will typically be a single word but is allowed to be a (system-specific) full filename specification; if the latter then the behavior is system-specific but normally path searching would be disabled.

Namearg should be the program name as it should be passed to the program. Generally, it should be exactly equal to progname. Hosts which do not pass a program's name to it should ignore it.

The *args* are the arguments to be passed to the program. Hosts which do not separate their arguments into words should concatenate the arguments back together with spaces between them. (Optionally, they could elect to not fully parse the line, and leave in the spaces as originally delivered.)

Note: The interpretation of the command may not be intuitive to an ordinary user of the host system; this interface is for precision, not intuition.

Exactly one of **CMD**, **EXEC**, or **GENERIC-CMD** must be present.

DIR *initial-directory*

Specifies (in a system-specific way) the initial default file system location for the program. If no **DIR** line is supplied the program is started in a system-specific initial location, presumably the user's ''home''.

Note: It is expected that this value would come from a source with a priori knowlege of the host - either the user or a ‘‘restart’’ request. It is not expected that an automated mechanism with no advance knowlege would be able to make use of this request.

MISC *registryname name=value*

Passes a value to the program using a system-specific mechanism. (Under POSIX and similar systems environment variables should generally be used.) The *registryname* specifies the naming authority, and is intended to prevent conflicts and allow for intelligent conversion. The idea is that systems that understand a given registry will map these straight into environment variables. Systems that don’t entirely understand a given registry or use a different but convertable mechanism can be picky and convert as needed. An appendix lists the names that all systems are strongly encouraged to support.

Note: The names in the ‘‘suggested’’ appendix are expected to be supplied by requesters with no advance knowlege of the host, only the blind assumption that the host will support those ‘‘well-known’’ names with their ‘‘well-known’’ semantics. Other names may be used by requesters with advance knowlege of the host - restart requests, for example.

Any number of **MISC** lines may be present.

DETACH

This line directs rstartd to attempt to ‘‘detach’’ the resulting process from the rsh connection, leaving as little overhead (processes, connections, etc) as possible. In POSIX-land, this will probably consist of redirecting standard input, output, and error to /dev/null or a log file and then executing the program using fork() and exec() and not having the parent wait.

It would be nice to have a mechanism for specifying where the output should be redirected - log file or perhaps log/display program/server. For the moment that’s left as a matter of local implementation and configuration.

Only one of **DETACH** and **NODETACH** may be present.

NODETACH

This line directs rstartd to attempt NOT to ‘‘detach’’. It is intended to allow one to override a configuration default to **DETACH**.

Only one of **DETACH**and **NODETACH** may be present.

AUTH *authscheme authdata ...*

Specifies authentication data to be used by the specified authentication scheme. This keyword may be given multiple times to give data for multiple authentication schemes or for a single scheme to use for multiple purposes.

System-specific lines begin with **SYSTEMNAME-**. No system-independent line will be defined that includes a "-" in its name. **X-** is reserved for experimental use. ("Reserved" is an interesting word, there; anybody can use **X-** for experiments and nobody can rely on there not being a conflict.) **INTERNAL-** is reserved for internal use by rstartd.

POSIX-UMASK *nnn*

Sets the POSIX umask to *nnn* (octal).

MSDOS-DIR *d:path*

Sets the current directory on drive *d:* to *path.* This differs from **DIR** in that **DIR** sets the current working drive and directory and **MSDOS-DIR** doesn't set the current drive; **MSDOS-DIR** would be used to set the current directory on drives other than the current one.

DESQVIEW-MEM *nnn*

Requests *nnn* kilobytes of conventional memory.

DESQVIEW-EXPMEM *nnn*

Requests *nnn* kilobytes of expanded memory. (To be more verbose, requests that the "max expanded memory" parameter be *nnn*.)

GENERIC-xxx

The **GENERIC** system is a hypothetical system that offers various services that can be supported on different real systems; it provides a common interface for heterogenous systems.

GENERIC-CMD *generic-command-name args ...*

Runs the local equivalent to *generic-command-name*. See the section on generic commands for more information. Exactly one of **CMD**, **EXEC**, or **GENERIC-CMD** must be present.

Generic Commands

Unless otherwise noted, generic commands are optional.

GENERIC-CMD Terminal

Runs a default terminal emulator for the current context. (It's not clear what, if anything, this means outside a windowing system context.) In POSIX X contexts this probably means xterm.

GENERIC-CMD LoadMonitor

Runs a default load monitor for the current context. In POSIX X contexts, this probably means xload.

GENERIC-CMD ListContexts

Sends to standard output a list of available contexts, one per line, with a comma-separated list of context names followed by a space followed by a brief description of the context. If multiple context names invoke the same context (as for instance X, X11, and X11R4 might) ListContexts SHOULD list the most specific context first.

This command MUST be available in the Default context, and SHOULD be available in every context.

GENERIC-CMD ListGenericCommands

Sends to standard output a list of generic commands available in the current context, one per line, with the command name followed by a space followed by a brief description of the command.

This command SHOULD be available in every context.

Contexts

A request can specify what *context* a program should be run in. A context will most likely include a set of default values for all of the controllable aspects of the program's execution environment.

Examples of Predefined Context Names

None	A minimal environment. Under POSIX, no environment variables.
Default	The default environment.
X	The X Window System, any version
X11	Version 11 of the X Window System, any release
X11R4	Version 11 of the X Window System, Release 4
X11R5	Version 11 of the X Window System, Release 5
OpenWindows	Sun's OpenWindows, any version
OpenWindows2	Version 2 of Sun's OpenWindows
OpenWindows3	Version 3 of Sun's OpenWindows
NeWS	Some version of Sun's Network Window System.

Contexts are allowed (encouraged even) to support multiple names for the same environment configuration. For instance: "X", "X11", "X11R4" might all refer to exactly the same configuration. "OpenWindows3" might well refer to a configuration that is a union of "X11R4" and "NeWS".

Suggested ‘‘required’’ MISC commands

General

For most of these the ‘‘simple’’ behavior is to simply pass the value to the app as an environment variable. This should be appropriate on any POSIX systems. Other systems should use whatever mechanism is appropriate to the given item; note that different mechanisms may be appropriate to different items.

All systems are encouraged to ‘‘understand’’ all of these names, translating them as appropriate to the local environment.

MISC X LANG=*locale identifier*

Specifies the locale desired. If POSIX specifies a list of locale identifiers then we can use that (and move it to the POSIX registry), but if not then we will have to define a list (perhaps based on a good de facto standard). Note that if there is no POSIX-standardized list of locale names the host SHOULD translate from our list to the local list. Note that this is based on the POSIX LANG environment variable, but rumor says that POSIX does not specify a list of identifiers; since we want a standarized list we must specify a registry and must assume ‘‘control’’ of the name. If in the future POSIX specifies a different list then during a transition period MISC X LANG and MISC POSIX LANG would perform similar functions but using different names for the locales. [[It may be that ISO country/language codes may supply a suitable registry. -- jb]]

MISC X DISPLAY=*host***:***displaynum*

Specifies the X server the app is to connect to.

MISC X SESSION_MANAGER=tcp/*host***:***port*

Specifies the session manager the app is to connect to.

MISC POSIX TZ=*time zone info*

Specifies the time zone the user is in, using POSIX standard notation.

Specific Notes on use of Extended rsh with the X Window System

To start an X program, the requester should specify an X context (“X” if nothing else) and specify the display parameter by saying

MISC X DISPLAY=*host***:***port.screen*

If the program is to join a session the requester should say:

MISC X SESSION_MANAGER=tcp/*host***:***port*

An X host MUST understand at least the X context and MUST, regardless of whether or not it has environment variables as a system concept, interpret **MISC X DISPLAY=***host***:***port* and pass it to the program in whatever way is locally appropriate. An X host supporting session management (which all will do by the time this is adopted, right?) MUST interpret **MISC X SESSION_MANAGER=tcp/***host***:***port* similarly.

An X host MUST understand the **X11** authentication scheme for the **AUTH** keyword. The data given MUST be in the form given by “xauth list” and understood by “xauth add”. AUTH X11 may be given several times to pass multiple kinds of authorization data or data about multiple displays.

Input methods? Extensibility? (Would be nice if new features could be incorporated without new rstartd executables, which argues for passing most data as environment variables.)

Suggestions for Implementation

Configurability, extensibility. Nobody is ever happy with the default. The host implementation should allow for as much configurability and extensibility as possible. In particular, provision should be made for administrator- and user- defined contexts, with user defined contexts overriding or augmenting the administrator-defined and system-supplied contexts. One good implementation scheme would be to have system-wide and per-user configuration files which use the same syntax as the protocol. ListContexts SHOULD list both system-wide and per-user contexts.

Provision SHOULD be made for administrator- and user- defined generic commands, with user-defined commands overriding system-wide ones. ListGenericCommands SHOULD list both.

Notes on Outstanding specification issues

Note: This is not part of the proposal.

Syntax

The syntax is OK so far for machine generation, but is yucky for human generation, especially spaces in environment variables in config files.

Environment variables

it would be nice if one could say things like $HOME in values, especially in config files.

Error handling

how to reliably mark successfully starting the app. Can report syntax errors or lack thereof, but reporting startup or execution errors is problematical. (Application execution errors are tough no matter what; startup errors are tough because you want to report an error if the exec*() fails, but if it succeeds you don't have the opportunity to give a success report.) [[Clive gives a clever trick for reliably distinguishing success, but I think there's a race condition if the app is allowed to use the rsh channel for output after it starts. -- jb]]

Error handling

how to report post-startup errors and warnings to the user. (Can say ‘‘not our problem’’, but it would be nice if there was a standard way to say ‘‘give output to a program that will report it back to the user’’ or something like that. Session Manager ‘‘death reason’’ messages might be adequate, but don't cover warnings.)

I18n

What character set should messages and requests be in? Can this be determined by the locale as specified by **MISC X LANG**? (Does this require that **MISC X LANG** be one of the first lines?)

Protocol

One reviewer suggests an FTP- and SMTP- style protocol with requests and responses. Personally I don't think individual responses to the lines are required and that the roundtrips involved would be wasted. The entire conversation is a single request and can be rejected or accepted en toto. The response mechanism does not provide for fabulously rich interpretation by an automaton, but it does allow for a success / serious failure / routine failure / warning distinction.

Multiple requests/connection

Some have suggested that you should be able to start multiple programs with a single connection, either to start multiple apps on login or to allow a ‘‘master’’ to request apps occasionally during the course of a session. As doing so would make it less possible (or at least more difficult) to use the data channel for communicating with the app after start, and as the overhead of starting a new connection isn't all that high, I'm not enchanted with the idea. Admittedly, if the overhead involved got higher (how fast is a Kerberos authentication?) it might become more attractive.

Conclusion:

A small protocol could be easily defined which would layer on top of a relatively primitive remote execution facility like rsh, rexec, or krsh, that would allow flexible application startup in a fairly machine-independent way. By mandating appropriate local configuration, X applications could be started (or restarted) conveniently, without requiring the requester to understand the detailed requirements of the remote system. The implementation cost, for both the requester and the "server", is small. Security issues are "not our problem", since they are all handled by existing protocols.

Security Considerations

Security is assumed handled by the underlying remote execution mechanism. The "helper" program described by this memo runs with the privileges of the user and so generally introduces no additional security considerations. Systems where security is controlled by controlling what programs may be run - where programs are trusted but users are not - may see a security impact unless their "helper" is careful about what programs it is willing to run.

Copyright

Author's Address:

Jordan Brown
Quarterdeck Office Systems
Santa Monica, CA
jbrown@qdeck.com