Remote Signaling in a Heterogeneous Unix Environment

J. Brothers and R. E. Newman-Wolfe

January 22, 1993

Abstract: This paper describes the rsig remote process signaling system. It operates in a heterogeneous network of Unix workstations and provides users the ability to issue signals across machine boundaries. Some of the efficiency and all the conceptual and practical convenience of signals are afforded, in a manner similar to remote procedure calls.

1 Introduction

A signal is a built-in method in Unix (and many other operating systems) to send event notifications to a process [Stevens 90][Leffler 89]. Both the process owner, and the operating system can send these signals to a process. Typically, the operating system signals that errors have occurred, or that asynchronous data has arrived. Most signals sent by users are generated to control process execution. This allows users to terminate errant processes, and handle other aspects of process maintenance.

1.1 Problem

While signals are very useful, they are limited by the fact that they, like procedure calls, stop at the limits of the machine. In the case of signals generated by the OS, this is not a problem. Seldom, if ever does a process on one machine cause an access violation, or a floating point exception on another (remote) machine. However, users typically use signals to terminate processes, or to inform a process of an asynchronous event in order to avoid polling overhead. There are four major times when this needs to occur.

- Termination of errant processes (endless loop, deadlock, etc.);
- Interruption of a process to inform it of an asynchronous event;
- Termination of processes in a distributed system;
• Termination of processes when the process table is full.

The rsig system is designed to meet all of these needs.

1.2 Existing Approaches

Currently, there are a few methods available to send signals to remote processes. Some distributed systems, such as the OMNI system [Drummond et al. 92], developed at the University of Campinas, Brazil, have facilities to send signals to remote processes started by the system, and processes in contact with the system. However this system is unable to send signals to processes unknown to it. At the time of this paper, there is scant information on the generality or features of the remote signalling facility within OMNI, and so we cannot do the OMNI system full justice regarding its capabilities.

Another way to send signals to remote processes is readily available, and well known. It requires the user to log into the machine, either through the network, or physically, in order to kill the process. This is a simple and inefficient form of remote signaling. However, in the third case, especially if the processes were owned by different people, some sort of access to the other uids would be required. In the fourth case, there would be no way to create another process if the process table was full. The only solution would be to reboot the machine, or wait for processes to terminate on their own. Generally, existing Unix networks do not support remote signaling.

2 Description

2.1 Design Theory

To implement remote signalling, the first requirement is a communications method. Since signals cannot be sent across a network, it is necessary to translate the signals into something that can. The obvious answer is to use a message, with information about the signal, the target process id (pid), and the user who generated the message. If a message is to be passed to a remote server, there must be some process waiting for or willing to accept the message. Once the message is received, it must be validated, to ensure secure access. Only then can the message be processed, and the appropriate code executed.

For the communications, the choices vary depending on what is available on the system. Most, if not all Unix systems support the Tcp/IP protocol as the network interface Many systems have higher level protocols that use Tcp/IP.

For the server, there are two major approaches. The daemon approach, where a dedicated signal server resides on the target machine, and listens for signal messages, and the individual client approach. In the
individual client approach, each client willing to receive signal messages would listen independently of the other processes on the same machine.

2.2 The rsig system

The rsig system was designed to be portable, simple to use, and secure. Two versions of the system have been developed. Version one uses Tcp/Ip as the communications protocol. Version two uses Sunrpc and the associated facilities. Both versions use a server (daemon) on each machine in the network. In version one, certain security problems must be handled that do not exist in version two. Because of this, version one was significantly more cumbersome and complicated, and very difficult to extend.

To solve the problems mentioned above, the system was designed with the following features:

- Process registration
- Password authorization
- Remote process listing
- Remote process signalling

Process registration is the ability of a client process to register itself with the local daemon. The process can supply a “logical name,” which will be unique on the machine. A remote user could specify the process name, instead of the pid of the remote process. The same cumbersome use of rsh is necessary to retrieve process ids across a network.

Password authorization is the ability of a user’s program to register itself, with not only a name, but with a password as well. This password could be told to anyone the user desired. By specifying the password in the signal message, anyone who knew the password could send signals to the process. While this might allow unauthorized access, the registering program is not required to supply a password.

Process listing is the ability of a client to retrieve a list of all registered processes on the remote machine. The server returns the list to the client, which then displays the list.

Remote signaling is the key feature of the system. The user executes a rsig_signal() function call, passing certain parameters, such as the remote host, process name, password, or signal number. The rsig_signal() function builds the information into a message, and sends the message across the network to the remote machine’s rsig daemon. The daemon then processes the message, validates it, and then executes it.

2.3 Design Goals

The rsig system was designed to meet three major requirements:
• Simplicity of the user interface;
• Security of the messages;
• Portability to as many platforms as possible.

Simplicity — The system has been designed to use a library of function calls, each of the form rsig_*(). The details of the message, and the communications protocols are intended to be hidden from the user. This was achieved much more successfully in version two than in version one.

Portability — The system was designed to be as portable as possible. Both version one and two have been successfully ported to the following architectures:

- Sun4 (SunOs 4.1.1 and 4.1.2) (Sparc);
- Sun3 (SunOs 4.1.1) (mc68020);
- Hp-Apollo (Hp-Ux 7.0) (mc68020);
- Dec 5000 (Ultrix 4.2) (mips).

Both of the communications methods constrain the portability, although in different ways. Tcp/Ip is standard on all, or nearly all Unix systems. However, it is not always implemented in the same fashion from system to system. There are also certain architectural differences between hosts that Tcp/Ip does not handle, most notably byte-order. Sunrpc is more standardized across the platforms on which it is available. It also has built-in libraries to handle byte-order, and other architectural differences. However, it is not available on all platforms.

The most significant problem with portability is the difference in signal codes between Bell Laboratory’s System V Unix, and Berkeley’s BSD Unix [Stevens 90]. Currently, the daemon will determine which signal the message is requesting. However, this is not a critical problem, since many of the key signal codes are shared between the two systems.

The security of version two was much easier to implement. Sunrpc offers two forms of authorization, Unix, and DES. DES authorization uses encryption to ensure the correct code is being used to generate the messages. This allows any code that uses the correct rpc call to be secure, without the need for root access, or any of the other limitations of version one.
2.4 Implementation

The rsig system uses sockets to handle interprocess communication. Version one used connectionless UDP sockets. This limited the maximum message to 8k of data, and did not ensure that the message arrived at the destination. Because of this, a system of acknowledgments was required to ensure the data was received. The system was never fully developed, and version one was never fully reliable. However, the system was reliable enough to ensure that the daemon never blocked if a message failed to arrive. Version two was fully reliable, since it used Sunrpc, and used the connection-oriented TCP sockets as the underlying communications medium. This also allowed messages of any arbitrary size to be transmitted.

Both versions one and two used similar message structures:

```c
struct rsig_msg {
    short type;  // type of message
    short flags; // for control and authentication
    int sig;     // signal to be sent (if a signal message)
    int pid;     // registration client or signal target
    short uid;   // user id of sender
    short gid;   // group id of sender (not in version one)
    char log_name[64]; // logical name associated with pid
    char passwd[64];  // password associated with pid
    int inet;       // inet address of sender
};
```

There are three message types in version one: Request messages, authentication messages and control messages. Requests were simply the specifiers of the functions listed above. Authentication messages were used to validate clients requesting a secure action. Control messages were used to acknowledge messages, and to transmit multi-message blocks of data from the daemon to the client. Version two used two remote procedure calls — rsig_secure_call_1() and rsig_insecure_call_1(). Additional information was provided to the server in the case of the secure call, but the interface was exactly the same. The insecure call was provided in case authentication requires a significant amount of extra time.

The daemon handled the following requests: registration, de-registration, process listing and signalling. For both version one and version two, registration and deregistration are identical. One message is sent and an acknowledgement is returned.

Process listing in version one required the use of Start-Of-Frame, and End-Of-Frame control messages. After the request was received, the daemon sent a Start-Of-Frame control message to the client. The information on each registered process was encoded into a message. The messages were delivered to the client one at a time. After all the messages were sent, the End-Of-Frame message was delivered. In version two, the messages were packed together into one block of data, and sent to the client. There was no need for control messages.
Signal messages in version one were the most difficult to transmit. The client transmitted an authorization request message, with the port number of a root port the client had opened. The daemon made a connection to the root port, at which time the client transmitted the signal message to the daemon via the new connection. In version two, since the security mechanism was built in, there was no need for a call-back. The message was transmitted using the secure remote procedure call.

### 2.5 Future work

Future versions of the system involve sharing the registration information among one or more of the daemons. Version three will have a master daemon, and several slave daemons. Whenever a process is registered on one of the slaves, the slave will send a copy of the message to the master. Similarly, the master will receive de-registration messages. Thus, the master will have a complete picture of all registered clients in the network. A user will be able to retrieve this complete listing, and act upon the additional information provided there. While this structure presents a single point of failure, there are methods to make the system more fault-tolerant [Goyer et al. 90].

Version four is expected to be fully distributed, with each daemon sending a copy of the registration and de-registration messages to all of the other daemons within the group. This will increase the fault-tolerance of the system as a whole, however it may increase the number of messages on the network by a significant amount. Later versions will be designed to handle a hierarchy of networks, probably using a master-servant relationship on each network, with the masters fully distributed with each other at the higher levels. This strategy could be extended to handle networks of arbitrary depth.

### 3 Conclusions

The rsig system is designed to meet the needs of a wide range of users. It is designed to be run on every machine in a network, providing a common, familiar interface to all users. It has applications for users, developers and administrators. It is simple to use, portable to most versions of Unix, and reasonably secure. It allows additional access, and has features not available from the kill() system call, or the current method of remote signaling. While the system is simple, it is designed to handle a specific set of problems efficiently and without difficulty.

It is our intent to use the rsig facility as part of a collaborative computing environment, the DCS distributed conferencing system [Newman-Wolfe et al. 91]. Here its primary use will be to provide notification to processes of asynchronous events, and to permit group based process control without requiring special privileges.
References


[8] S. J. Leffler \textit{The Design and Implementation of the 4.3 B SD UNIX Operating System}. pp. 94-102, Addison-Wesley, Reading, Massachusetts

