Remote Signaling in a Heterogeneous Unix Environment

J. Brothers and R. E. Newman-Wolfe

January 22, 1992

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]. 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. Several distributed systems, such as the OMNI system, developed at the University of Campinas, Brazil, have facilities to send signals to remote processes started by the system. 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 individuals uid 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, 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 Tcplib protocol as the network interface [Comer 90]. Many systems have higher level protocols that use Tcplib.

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 Tcplib 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 process id (pid) of the remote process. Process ids require the same cumbersome method as remote signaling to retrieve across the 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 does not have 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 can return the list back to the client, who then displays the message.

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.

There is one major problem with porting, and that is the difference in signal codes between Bell Laboratory’s System V Unix, and Berkeley’s BSD Unix [Stevens 90][Leffler 89][Bach 86]. 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.

Security is the most crucial aspect of the system. Security under bare Tcp/Ip is very difficult. The only information that the protocol supplies about the remote host is the internet address. This makes authorization of the remote process and user a difficult task, since nothing in the message itself is ensured to be accurate. Version one followed the “root port” model used by rsh and other Unix utilities to remain secure. While this worked, it presented certain limitations to the functions the client could access. All secure requests had to be processed through a standalone program, and could not be used correctly while in a library.

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

Insert here - daemon/client binding, deamon/client responsibilities, message types, list of functionality, how functionility implemented,

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 informa-
tion provided there. While this structure presents a single point of failure, there are methods to make the system more fault-tolerant [Goyer 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


