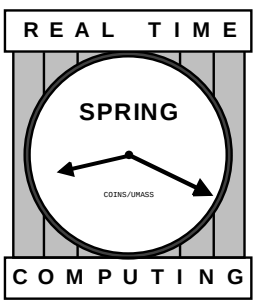
Spring Project Development Environment Target Node Software

Chung-Huei Kuan, Douglas Niehaus

Spring Project
Computer and Information Science Department
University of Massachusetts Amherst, Ma 01002

June 1990

Draft



Spring Project Documentation

Project Directors

Prof. K. Ramamritham

Prof. J. A. Stankovic

1 Introduction

The *Spring Target Node Software* is a collection of codes that is built upon the Spring Communication Facility[1]. The Spring Communication Facility is a library of primitives supporting a client/sever model of communication between two machines. On one machine, a *process server* functions as a front-end for a family of services. Upon receiving a request from a client running on the other machine, the process server invokes an appropriate program serving the client request. The Spring Target Node Software follows this style of communication and consists of the following portions:

- A program, `spr_server`, which, when invoked on the Spring target machine (i.e., the Motorola boxes), will run as the process server for the Spring Service Family. There are also two command scripts, `startsvr` and `stopsvr`, for starting and stopping the process server.
- A number of programs that constitute the Spring Service Family. Each program provides a specific service and is invoked by the `spr_server` program when it receives a client request for that service. When a service program is invoked, it is given a *socket descriptor* with which it will directly communicate with the client. Currently the two most important service programs are:
  - `loader`. This program accepts a S-record file or a Spring executable file, deciphers the file as the content of the file is received, and put the information contained in the file to the specified memory locations.
  - `rwer`. The client specifies a chunk of memory locations and the type of operation (either read or write). This program then either sends the contents of the specified memory locations to the client or stores the client input to the specified memory locations.
- A number of demonstration programs showing how to use the Spring Service Family.
- A service library containing some useful primitives. To communicate with the service programs, some protocols have to be adhered to. The primitives contained in this library provide a convenient interface for such communication. The library also contains some primitives used by the `loader` and `rwer` programs. People could use these primitives to build their own service programs.

This document is organized as follows: In Section 2, we will describe how to start the process server on a target machine so as to make various services available to processes running on the development machine. We will also mention some issues that are vital to the process server's

correct operation. Then, in Section 3, how to add new services to the Spring Service Family is described. In Section 4, the primitive contained in the service library are described. At last, in Section 5, the manual pages of various commands (including the demonstration programs) are included.

2 Spring Server

All the Spring Target Node programs and documents are under SCCS control with the unit name `server_stuff/spr_server`. To compile and install the Target Node Software on a target machine, first you have to populate them on a development machine and then use the command `make rtl?_pop` to copy everything to a target node's `/usr/local/spring/server_stuff/spr_server` directory. After successful installation, all you have to do is to issue the `startsvr` command on the target machine to start the Spring process server[1]. There is anther command, `stopsvr`, for stopping a running process server. Please see Section 5 for more details about these two commands. Since the Spring Target Node Software is built on top of the Spring Communication Facility. So before compiling and installing the Spring Target Node Software (as described above), make sure that the Spring Communication Facility is already installed. The Spring Communication Facility is under SCCS control with unit name `server_stuff/server`.

Several files located in the target machine's `/spring/etc` directory are very important to the Spring process server:

- `local_services`: This is the `service file` described in [1]. The process server consults this file at its startup phase in order to know what service progam to spawn for what service request. The current content of the file is as follows:

```
download      3002/0//usr/local/spring/loader
hello         3003/1//usr/local/spring/greeter
readwrite     3004/2//usr/local/spring/rwer
activate      3005/3//usr/local/spring/trigger
```

- `server_id`: This file contains the running process server's process ID such that the `stopsvr` command knows which process should be stopped[2]. On the other hand, the file is also used as a semaphore. When the process server starts to run, it will first check whether the file `server_id` already exists. If no, the process server will continue to run. Otherwise, the process server will assume that there is already a process server running and terminate itself. Since the `stopsvr` and `stopsvr` will always remove a "dangling" `server_id` file, a user shouldn't notice the existence of this file under normal circumstances.

[1] Please make sure that `/usr/local/spring` is in your `PATH`.
[2] Actually `stopsvr` simply sends a signal to the process whose ID is specified in `server_id` file.

- `server_log`: This is the process server's log file. If there is anything wrong with the process server, the logging information contained in this file may provide some clues. Similarly, almost all service programs have their own log files. For example, `loader`'s log file is `loader_log` and `rwer`'s log file is `rwer_log`.

3 Server/Service Interface

The process server on a target machine functions as a front-end which spawns appropriate service programs upon user requests. Currently the Spring Service Family has already contained several such service programs. New service programs can be easily added to the existing Spring Service Family, since the process server has covered almost all interconnection details and the Spring Communication Facility provides a number of useful primitives.

To add new service program to the Spring Service Family, the following conventions have to be observed:

- The service file `local_services` provides a consistent view of the available services among the clients and the process server. Please follow the file format described in [1] to register your new service(s). Also make sure that both the development machine and the target machine have an identical copy of the service file.

- In the service file, you have to specify the absolute path to your service program. So before using the new service program, make sure that a copy of the new program is at the specified place.

- Once a service program is spawned, it runs independently with the process server. The process server wouldn't wait for the spawned service program to complete and the spawned process can do whatever it sees fit. The only connection between the two is that the process server will pass a *socket descriptor*, a *semaphore number*, and a *semaphore ID* to the spawned process. The spawned process can extract these parameters as follows:

  ```
  sd = atoi(argv[1]);
  semno = atoi(argv[2]);
  semid = atoi(argv[3]);
  ```

  The spawned process can use the socket descriptor to communicate with the client immediately, and use the allocated semaphore for mutual exclusion. If you don't need to use the semaphore, you can just ignore it.

4 Service Library

• Target Side

```
FILE *Log_fp;

void
open_log(sd, this_file)
int  sd;
char *this_file;
```

This function opens a log file whose name is specified by `this_file`. If there is someting wrong when opening the file, this function will either terminate the caller immediately or, if `sd` is not equal to zero it will treat `sd` as a socket descriptor and tell the process at the other end of the socket that the communication cannot proceed.

```
void
log(logstr, timestamp)
char  *logstr;
int  timestamp;
```

This function writes a string of character pointed to by `logstr` into the log file opened earlier by a call to the `open_log` function. The parameter `timestamp` is a flag that, when set to one, will cause a timestamp to be attached to output string.

• Both Development and Target Sides

```
int
fp_to_socket(sd, memp, len)
int  sd;
FILE *memp;
unsigned len;
```

This function reads from the file specified by the file pointer `fp` for up to `len` bytes and sends them to the socket specified by `sd`.

A negative returned value indicates error. If the value is greater than -100, the error is related to the system call `send`. Otherwise it is related to the system call `fread`.

```
int
socket_to_fp(sd, memp, len)
int  sd;
FILE *memp;
unsigned len;
```

This function reads `len` bytes from the socket specified by `sd` and writes them to the file specified by the file pointer `fp`.

A negative returned value indicates errors. If the value is less than -200, the error is related to the function `Exact_Recv`[1]. Otherwise, it is related to the system call `fwrite`.

- **Development Side**

```
int
send_srec_file(fp, sd)
FILE     *fp;
int       sd;
```

This function sends the content of the file referenced by file pointer `fp` to the server associated with the socket descriptor `sd`. The file is assumed to consist of records separated by newline characters whose length (excluding the newline character) is not longer than MAXBUFF.

This function uses the same socket descriptor `sd` both for sending s-records and receiving UNACK messages from the receiver. It uses the system call `select` to determine if the socket is ready for writing or there is a message from the receiver. If nothing happens in TIMEOUT seconds, the procedure return with a value RCVTIMEOUT (=-103). If an UNACK is received, RCVRUNACK (= -101)is returned. If there is something wrong with the system call `send`, -1* errno is returned. If everything is OK, 0 is returned.

```
int
send_exec_file(fp, sd)
FILE *fp;
int  sd;
```

This function sends a Spring executable file referenced by the file pointer `fp` to another machine via a socket descriptor `sd`. It will push the loading address and the image size into the socket before the binary image is actually sent. The loading address and the image size is deciphered from the executable file.

If nothing happens in TIMEOUT seconds, RCVTIMEOUT (= -103) is returned. If UNACK message is received, RCVRUNACK (= -101)is returned. If there is something wrong with the system call `send`, -1* errno is returned. If there is something wrong with fread, RCVEOF (=-102) is returned. If everything is OK, 0 is returned.

5 Manual Pages

References

[1] Chung-Huei Kuan, Spring Project Development Environment Communication Facility, *Spring Project Documentation*, June 1990.