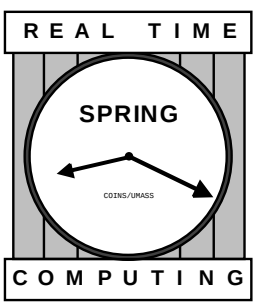
Spring Project Development Environment Communication Facility

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

This document describes a set of *primitives* designed for the Spring project's long term use. Based on the BSD *socket* mechanism, this set of primitives support a client/server model of communication between a process running on a Spring development machine and a process running on a Spring target machine. Due to the current structure of the Spring development environment, the primitives assume that the client process is running on Ultrix[1] while the server process is running on System V UNIX[2]. However, with minimum modifications, these primitves can be compiled and run on other UNIX compatible operating systems, as long as the socket mechanism is supported. Besides the specific style of client/server communication, the primitives in this communication library can be used for general inter-process communication using sockets.

This document is organized as follows: In Section 2, an overview of the design is given. Then, we divide the primitives in the communication library into three categories and describe their usage in three separate sections. In the last section, the primitive interfaces are described.

2 Design

This set of primitives was originally a part of the project for automating the downloading of Spring images (in S-record format) from a development machine to a target machine. Because of its flexible design, it was then used to build the Spring Target Node Server which, in addition to downloading, supports various services such as reading/writing target machine memory and reseting target machine etc. For more details about the supported services, please refer to [1].

When these primitives were developed, we decided that they should not be just procedural abstraction of large chunks of code, but should be designed to support a style of communication with which various interactions between the development machine and the target machine can be built easily. The style of communication we chose is very similar to the *client/server model*[2] used in the APARNET, in which a *process server* or *logger* running on the target machine listens to a set of *ports* for connection requests while, to request a particular service from the server, a client process running on the development machine has to connect to a specific port monitored by the server. Based on the port where a connection request is from, the process server spawns an appropriate process for handling the client's request. The advantage of this style of communication is that the glory details of interconnection are hidden inside the process server and concealed from the service program developers.

The set of services provided by a process server is said to consist of a *service family*. To facilitate

[1] Ultrix is a registered trademark of Digital Equipment Corporation.
[2] UNIX is a registered trademark of AT&T.

the process of adding or removing services from an existing service family, we associate each process server with a *service file* in which information regarding each service within the service family is kept. The service file is used both by the process server and the clients. A client consults the service file to find out the port number of a particular service and the process server uses it to post the services to the outside world and decide which program is responsible for which service.

For each supported service, a single line should be present in the service file with the following information:

1. a *service name*,
2. a *port number* with which the service is associated,
3. a *semaphore number* for mutual exclusion, and
4. a *service program name* by which the service is handled.

The line format supported by this set of primitives is shown as follows:

```
download  3000/0 loader
```

The first ten characters contain the service name(with padded blanks, if necessary). The number starting from the 11th position and before the slash is interpreted as the port number and the next number between the slash and a blank is the semaphore number. The remaining characters (up to the terminating newline character) contain the name of the service program.

3 Service File Access

These primitives always take a file pointer to an open service file and extract information from it. A brief summary of the primitives are as follows:

- `Get_All_Services` reads the entire service file into an array of structures supplied by the caller. Each element of the array contains all the information about a service.
- `Find_Service` puts the information about a specific service into a caller-supplied structure.
- `Find_Port` returns the port number of a particular service.
- `Find_Server` finds the name of the service program that handles a particular service.

For more detailed description about the parameters and returned values, please see the last section.

4 Client/Server Connection

Almost all primitives within this category are designed for the process server. Function `Setup_Internet_Pipe` is the only exception. Given the name of a remote host and a port number, this function will set up a *stream socket* to a particular service program running on the remote host. A common scenario for a client to request service from a remote server can be described as follows: The client first opens the service file, uses the `Find_Port` function to get the port number associated with the requested service, then sets up a socket using the `Setup_Internet_Pipe` function. When the function successfully returns, the client can be sure that a service program is ready for request at the other end of the socket.

Among the rest of the primitives, the `Post_All_Services` primitive and the `Multiplex_Services` primitive are the most important ones. The `Post_All_Service` primitive reads a service file and then

1. creates a socket for each supported service,
2. binds each socket to a particular port as specified in the service file,
3. informs the kernel of its willingness to accept connections from these port.

The `Multiplex_Services` uses the `select` system call to monitor the set of sockets created by the `Post_All_Services` primitive until some connection request is available at some socket or a maximum interval for selection to complete has expired. Whenever a connection request is received, the `Multiplex_Services` `accepts` it, obtains a new socket for this connection, `forks` and `execs` the socket's associated service program, and passes the new socket (among other things) to the forked process. After forking the service process, the process server goes back to check for new connections without waiting for the server process to terminate.

5 Synchronization

An important problem of our model of communication is the synchronization between the process server and its child processes. For greater efficiency and flexibility, the process server shouldn't wait for the spawned server to complete before responding to the next connection request. If this restriction is released, however, it is possible that more than one child process for the same service could exist simultaneously (which may not be desirable). The communication library provides a few primitives which use semaphores to support such a mutual exclusion requirement.

`Get_Sem_Set` primitive obtains a set of semaphores from the kernel and initializes them to a user-supplied value. Depending on the initial value used, these semaphores can be used as counting

semaphores or binary semaphores. This set of semaphores is identified by a single ID. To access one of the semaphores, both the set ID and the semaphore's number have to be specified. The `Server_P` and the `Server_V` primitives provide the conventional `P` and `V` operations. Another primitive `Rm_Sem_Set` is used to remove the entire set of semaphores. All these primitives will not be blocked when the desired operations cannot be completed but return an error code.

6 Command Interface

Our model of communication imposes no constraint on how the client should talk to its service program and vice versa. The connection established between the client and the server is a *reliable, in-order, bi-directional byte stream* which has no message or record boundaries. The designer of a service program can implement whatever protocol he sees fit, as long as the client process follows the same protocol. However, this communication library supports a simple *command* interface. A *command* consists of a command code and two object strings (up to a certain number of bytes). The C type definition for the command structure is as follows:

```
struct command {
        char codename;
        char obj1[MAXNAME];
        char obj2[MAXNAME];
        };
```

The two strings specify what object(s) this command is going to affect. The `Send_Command` primitive takes a character (code) and two strings, arranges them into a byte array, and sends the byte array out through a socket. On the server site, the `Get_Command` primitive reconstructs the received byte array into a command structure.

These two primitives are built upon another two primitives, which may be useful for other applications. The `Nonblock_Recv` primitive reads data from a socket but, unlike the `recv` system call, will be blocked only for a fixed amount of time if no data is available. The `Exact_Recv` primitive is also non-blocking and will return successfully only if a user-specified number of bytes is received from a socket before timeout expires.

7 Primitive Interfaces

• Mutual Exclusion

```
int
Get_Sem_Set(key, nsems, initval)
int     key;
int     nsems;
int     initval;
```

This function gets a set of `nsems` semaphores identified by `key` from the kernel and sets all the semaphore values to the `initval`. The ID of the set of semaphores will be returned upon successful completion. If something is wrong when calling the `semget` and `semctl` functions, a negative value would be returned. The meaning of the value is as follows:

- $-1 * errno$: when the error is due to semget
- $-100 + -1 * errno$: when the error is due to `semctl`
- $SEMEXIST(-201)$: if the semaphore already exists.

```
int
Rm_Sem_Set(semid, nsems, initval)
int semid;
int     nsems;
int     initval;
```

This function is the reverse of `Get_Sem_Set()` function. It'll remove the semaphore set `semid` if each semaphore in the set (of size `nsems`) is equal to its initial value `initval`. If there is something wrong with the `semctl` call, a negative value $-1 * errno$ will be returned. If `semctl` is OK, but some semaphore in the set is not equal to its initial value, a negative value $SEMALTERED(= -101)$ is returned.

```
int
Server_P(semid, semno)
int semid;
int semno;
```

This function performs non-blocking semaphore P operation on the semaphore `semno` within the semaphore set `semid`.

It will return zero if P operation is OK (i.e., the semaphore value is larger than 0). Otherwise, a negative value is returned. If it is because of the `semop` call, the returned value is equal to $-1 * errno$. If `semop` is OK but the semaphore value is less or equal to 0, a value $SEMNEG(= -101)$ is returned.

```
void
Server_V(semid, semno)
int semid;
int semno;
```

The inverse of the `Server_P()` function.

- **Service File Manipulation**

```
int
Get_All_Services(fp,ser)
FILE   *fp;
struct service ser[];
```

This function reads all the service records in the service file specified by the file pointer `fp` into an array of service structure `ser[]`. The file format is as follows:

1. Each service is in a separate line.
2. The first field is the service name (up to 10 lower case letters).
3. The second field is the port number.
4. The third field is the semaphore number.
5. The fourth field is the file name that will be executed when the corresponding service is requested.
6. Field 2,3, and 4 are separated by `/`.

This function returns the number of service records read. The user should make sure that there is enough space in `ser[]` for storing the records.

```
int
Find_Service(fp, ser, rec)
FILE   *fp;
char   *ser;
struct service *rec;
```

This function examines the service file (as specified by the file pointer `fp`) to find the service record about the service whose name is `ser`. The found service is returned in `rec`.

This function returns -1 if the specified service is not found. Otherwise 0 is returned.

```
int
Find_Port (fp, ser_name)
FILE   *fp;
char   *ser_name;
```

This function examines the service file (as specified by the file pointer `fp`) to find the port number associated with the service `ser_name`.

This function returns the port number found or -1 if the specified service is not found.

```
int
Find_Sem (fp, ser_name)
FILE *fp;
char *ser_name;
```

This function examines the service file (as specified by the file pointer `fp`) to find the semaphore number associated with the service `ser_name`.

This function returns the semaphore number found or -1 if the specified service is not found.

```
int
Find_Server (fp, ser_name, name)
FILE *fp;
char *ser_name;
char *name;
```

This function examines the service file (as specified by the file pointer `fp`) to find the server program associated with the service `ser_name`. The name of the program is returned by `name`.

This function returns -1 if the specified service is not found.

- **Client/Server Connection**

```
int
Post_Service(port_no)
int port_no;
```

This functions returns a socket (descriptor) to the port `port_no` via which a service is provided. If there is anything wrong, a negative value will be returned. If the value is

- > -100, the error is related to `socket`
- > -200, the error is related to `gethostname`
- > -300, the error is related to `gethostbyname`
- > -400, the error is related to `bind`
- > -500, the error is related to `listen`

```
int
Connect_to_Client(original_socket, client)
int    original_socket;
struct sockaddr_in *client;
```

Set up a socket from which the client's order is received. If socket is created successfully, the socket descriptor is returned. The information about the connected client is also returned. If there is anything wrong, a negative value ($-500 - errno$) is returned.

```
int
Post_and_Connect(port_no, client)
int port_no;
struct sockaddr_in *client;
```

This function is a combination of the previous two functions. It would be useful for setting up a server that offers only one type of service.

One major difference between using `Post_and_Connect` and `Post_Service` + `Connect_to_Client` is that the former will be blocked until a connection with a client is set up. Also the former hides the detail that a temporary socket is created for accepting connection request, whereas the user of the latter one has to close the temporary socket himself.

If there is something wrong, a negative value will be returned. If the value is

- > -100, the error is related to `socket`
- > -200, the error is related to `gethostnam`

- > -300, the error is related to `gethostbyname`
- > -400, the error is related to `bind`
- > -500, the error is related to `listen`
- > -600, the error is related to `accept`

```
void
Post_All_Services(services, n)
struct service services[];
int    *n;
```

This function simply calls the `Post_Service` function repeatedly to post all the supported services. If everything is OK, the value 0 is returned. Otherwise a negative value is returned. For the meaning of the value, see `Post_Service`. And the number of service record successfully read before some error occurs is contained in `n`.

```
void
Shutdown_All_Services(services, n)
struct service services[];
int    n;
```

This function informs the kernel that the caller has no intent to either read or write data from or to the sockets associated with the `n` services contained in the array `services`.

```
int
Multiplex_Services(services, n, semid, timeout, ser)
struct service services[];
int n;
int     semid;
int     timeout;
struct  service *ser;
```

This function monitors a set of sockets for connection requests. It then forks appropriate server process for serving client's request and returns zero to the caller. The pointer `ser` points to a structure specifying the requested service.

If no connection request has been received for a timeout amount of time, this function also returns to the caller with return value 0 but with a null `ser`. The timeout value should be in seconds.

If an error occurs, a negative value is returned and the meaning of the value is as follows: If the value

- > -100, the error is due to `select`,
- > -200, the error is due to `accept`,
- > -300, the error is due to `fork`,
- $= EXECERR(-301)$, the server cannot be executed.

A pointer to the particular service that causes the error is also returned. If the error is not related to any service, a NULL pointer is returned.

```
int
Setup_Internet_Pipe (remote_host, port_no)
char remote_host;
int port_no;
```

Create a Internet stream to a remote server process on the node `remote_host` and whose service is identified by `port_no`.

If anything is wrong in creating the stream, this function returns a negative value with the following meaning: If the value

- > -100, the error is due to `socket`,
- > -200, the error is due to `gethostbyname`,
- > -300, the error is due to `connect`.

If the stream is established successfully, the socket descriptor is returned.

- **Command Interface**

```
int
Get_Command(sd, op, to)
int  sd;
struct  command *op;
int to;
```

This function reads input from a socket specified by `sd` and subsequently deciphers it into a command structure pointed by `op`. This function will be blocked no more than `to` seconds. The error code is the same as `Exact_Recv`.

```
int
Send_Command(sd, op, obj1, obj2)
int  sd;
char op;
char *obj1, *obj2;
```

This function sends a command through socket `sd` with a message format as follows: (1) the first character is the command code (`op`) specifying the the requested operation, (2) the second byte specifies the length of the object string `obj1` (the length includes the trailing '0'), (3) the object string `obj1`, (4) The second object string `obj2` has the same format as (2) and (3).

A negative returned value means something is wrong. If the negative value is > -100, the error is due to `send` function. If the value is equal to $OBJTOOBIG(= -101)$, this means `obj1` and/or `obj2` are too long.

```
int
Exact_Recv(sd,buff,n,to)
int  sd;
int n;
char *buff;
int to;
```

This function get exact number (`n`) of bytes from a stream socket (`sd`) and store them in a buffer (`buff`). If the number of bytes available is not enough, the caller will be blocked at most `to` seconds. This function returns the same error code as `Nonblock_Recv`.

```
int
Nonblock_Recv(sd, buff, len, waittime)
int sd;
char *buff;
int len;
int waittime;
```

This function reads from a stream socket (`sd`) and will be blocked for a fixed amount of time (`waittime` seconds). Upon successful completion, the number of bytes received will be returned. A negative returned value means that some error has occured. If the returned value is > -100, the error is due to `select`. If the value > -200, the error is due to `recv`. If the returned value is equal to 0, the function returns because of timeout.

References

[1] Chung-Huei Kuan, Spring Project Development Environment Target Node Software, *Spring Project Documentation*, June 1990.

[2] Andrew S. Tanenbaum, *Computer Networks*, pp387–391, Prentice-Hall, 1988.