Multiprocessing, Semaphores and Networking with ACL
SNeRG Technical Note 33

John F. Santore
Department of Computer Science and Engineering
and SNePS Research Group
University at Buffalo, The State University of New York
Buffalo, N.Y. 14260-2000
jsantore@cse.buffalo.edu

5th September 2002

1 Multiprocessing in ACL in Brief

When Franz uses the term "multiprocessing", they mean the use of multiple Lisp processes or threads in the same Lisp world. (This contrasts with the definition of multiprocessing as the coordinated processing of a single program using more than one processor, a definition that many computer scientists use.) When you use multiple processes for your program, you assign some of your programming tasks to another execution thread, which will then run separately from the main read-eval-print loop. Theoretically the multiple processes could run on more than one processor. However, as of ACL 6.2 the UNIX version of the ACL multiprocessing package does only simulated multi-threading.

ACL provides the `multiprocessing` package, which contains the functions and data structures necessary to run your program in multiple Lisp processes. The `multiprocessing` package is nicknamed `mp`. I'll use this nickname when referring to it below.

In order to use the multiprocessing methods, make sure the multiprocessing package is loaded with

```
(if (not (find-package :mp))
    (require :process))
```

When you want to run a part of your program in a separate process, use the function

`(mp:process-run-function` *name-or-keywords function &rest args*`)`.

This function will take care of creating the process and then running it in its own execution thread. It will return the process object that it creates. If you give `mp:process-run-function` the name of the new process as a string for the first argument, that name is for you the user to be able to identify the process later. If execution completes in the new process, then the process will be killed.

Below is an example of creating a new process to run a function. The name of the new process is '`Language thread`', and the function that the process will start running is named `ProcessHearingInput`. This function takes no parameters.

```
(mp:process-run-function "Language thread" #'ProcessHearingInput)
```

If you find one of your processes that you started has gotten into an infinite loop, or for some reason you want to remove it before it would normally terminate, you can do so with the function `(mp:process-kill` *process*`)`. You can either get the process from the return value of your previous call of `mp:process-run-function` or you can get the process out of the list of processes stored in the `mp:*all-processes*` variable.

2 Semaphores in Lisp

2.1 Introduction

Whenever there is more than one process in a program, one has to worry about making sure that the different processes do not mangle their shared resources. This is because each process will run for a "time slice" which is determined by a scheduler, which might be part of Lisp or part of the operating system. The scheduler determines when one process stops running and another process starts running. The scheduler can make this decision at any time. If we are not careful, this can lead to corruption of data when two processes try to update the same data. As a motivating example, consider a global variable that holds the number of foo objects created. Every time a new foo is created, the counter must be incremented by one. If there is no protection in place and we have two processes creating foo objects, we might see this scenario. Process one is going to create a foo, and so it gets the current value of the count variable, which is 6, and adds one, for a value of 7. Before process one can set the count to this new value, the scheduler decides that it is time for process two to run. Process two wants to create a new foo, so it gets the current value of the count variable which is still 6. It adds one, for a total of 7, and writes that value to the count variable. Later when the scheduler decides that it is time for process one to run again, it continues where it left off, it writes the value of 7 to the count. So now the count of foo objects is incorrect because both processes updated the count using the same old value. Clearly we need some way of preventing this from happening.

Semaphores are used in multi-threaded systems to protect shared resources such as global variables or data structures. You can usually assume that assignment is an atomic operation (By which I mean that one process will be guaranteed to complete a simple assignment without the scheduler running another process in the middle of the statement). However, retrieving the value of a variable, and then updating it with assignment, is not atomic. Also doing something like writing to a stream in Lisp is not atomic. Both of these fall under the heading of the "mutual exclusion" problem. If there are no checks, the output of two programs or threads can accidentally intermingle and become garbled.

Because semaphores will cause the execution to block for a process that is waiting on a semaphore that is in use, you really need to use semaphores only to protect the code that absolutely has to be protected and not any extra code.

2.2 Using Semaphores in ACL

Semaphores are called "process-locks" in ACL. I will provide examples of creating and using semaphores with ACL code. These examples show the use of a macro to use the semaphore. This macro, `with-process-lock`, is a simple way to use semaphores in ACL. It has worked for all of my needs so far.

All of the semaphore functions are in the `mp` (multiprocessing) package. The `mp` package provides all of the multi-threading capabilities of ACL as of v6.2

2.2.1 An annotated example of creating and using a semaphore in ACL

Suppose you have a global variable that is incremented every time we create a new CLOS (Common Lisp Object System) object from the "foo" class. Lets call it `*fooCount*` and initialize it to zero:

```
(defvar *fooCount* 0)
```

In order to use a semaphore in your code, you must first declare it and create it. For example:

```
(defvar *fooCount*Semaphore (mp:make-process-lock))
```

Here we are using the `make-process-lock` function from the `mp` package to create a semaphore which will be assigned as the initial value of our new global variable `*fooCount*Semaphore`.

Now every time we create a foo object we need to increment the count. Because incrementing a variable is not atomic, we should protect this action with our semaphore. (There should be one semaphore for every task, global variable or global data structure that will be modified of course. You cannot reuse `*fooCount*Semaphore` for something else if we are using it to protect access to `fooCount`.)

To protect access to `*fooCount*`, we will use the `with-process-lock` macro.

```
(mp:with-process-lock (*fooCount*Semaphore)
   ;;put the body of your task here
   ;;so for our example, using the incf macro
   (incf *fooCount*)
)
```

Here the call to `mp:with-process-lock` takes the semaphore you need to use for this task (`*fooCount*Semaphore` in this case) and in the body, you place the statements that should be protected.

If your process throws an error from within the body being executed from within an `mp:with-process-lock` block, the process will abort and the semaphore will be released properly. Your remaining processes will be able to use the semaphore without a problem.

Figure 1 is Professor Shapiro's "eager beaver and" macro which uses both multiprocessing and semaphores in a short but realistic example of both.

See Franz's documentation[1] on this topic for more information.

3 Communicating between SNePS (running in ACL) and another program using sockets.

3.1 Introduction

We want our SNePS program to communicate with some other program running on another computer, and possibly written in another programming language. In order to do this we need to use a standard method of communicating between computers. ACL does support OLE as a method of communicating between programs, but that is a Microsoft-ism and does not work well on all platforms. I've chosen to use standard sockets because every modern operating system makes them available. In this section of the document, I'll discuss setting up socket communication between Lisp and other programs and then illustrate the discussion with some examples of connecting the SNePS natural language features to another program.

The basic socket interface was designed in C for the BSD flavor of UNIX. It is quite arcane, so many programming environments have libraries or packages with their own socket interface (implemented as wrappers around the original interface) that are easier to use. The ACL socket interface is found in the `acl-socket` package, nicknamed `socket`. I'll use this nickname in my examples.

Sockets are defined by the combination of one machine and a port on that machine on one end of the connection. A socket connection is defined as a socket on one machine being attached to the socket on another machine

When working with sockets, one program must act as the server, while any number of other programs can act as clients. The server starts first, and waits for clients to connect. The server must have a known machine name and port number. The client then connects to the socket on that machine with that port. The machine and port of the client need not be explicitly known to the user; once the client connects to the server process, the server process will know who it is talking to. Once the connections are made, the client and server processes can communicate by sending data over the sockets.

In ACL, once you open a socket connection between the client and the server, the connection is represented as a stream. You can read and write to this stream like any other Lisp stream. However, when reading from the socket, it might be better to wait until there is something to read. You can use the `mp:wait-for-input-available(streams &key wait-function whostate timeout)` function to do this. I'll demonstrate its use in the example below.

3.2 Annotated Example

In the example below, I'm assuming that the SNePS/ACL program is acting as the client, and the other program will act as the server. The ACL side could run as the server, but in my demos, I have not done it this way so I'll talk about what I know works.

```
(defmacro eband (&rest args)
  "Evaluates its arguments in parallel, in separate processes.
  Returns True if all its arguments evaluate to True.
  Returns nil if, and as soon as, any of its arguments evaluate to nil."
  ;; Written by Stuart C. Shapiro
  (let ((flag (gensym))   ; to get final answer
    (flagSemaphore (gensym))
    (childProcessCount (gensym)) ; number of child processes
    (childProcessCountSemaphore (gensym))
    (localresult (gensym)))  ; local to each child process
    `(let ((,flag t) (,flagSemaphore (mp:make-process-lock))
      (,childProcessCount ,(length args))
      (,childProcessCountSemaphore (mp:make-process-lock)))
      ,@(mapcar ;; Spawn a child process for each argument.
        #'(lambda (arg)
          `(mp:process-run-function
            (list
              :name (format nil "Child for ~A" ',arg)
              :resume-hook
              ;; Evaluated whenever process is resumed.
              #'(lambda ()
                (when (null ,flag)
                  ;; As soon as flag is set to nil,
                  ;; decrement count and quit.
                  (mp:with-process-lock (,childProcessCountSemaphore)
                    (decf ,childProcessCount))
                  (mp:process-kill mp:*current-process*))))
                ;; Function run by each child process.
                #'(lambda ()
                  (format t "~&Processing ~A" ,arg)
                  (let ((,localresult ,arg)) ; Here's what takes time.
                    ;; Contribute the value of its arg
                    ;; to the final value.
                    (mp:with-process-lock (,flagSemaphore)
                      (setf ,flag (and ,flag ,localresult)))
                    ;; When finish, check out.
                    (mp:with-process-lock (,childProcessCountSemaphore)
                    (decf ,childProcessCount))))))
                args)
              ;; Wait until some child process sets the answer to nil,
              ;; or until all child processes are done.
              (mp:process-wait
                "Checking wait status"
                #'(lambda ()
                (or (null ,flag)
                (zerop ,childProcessCount))))
            ;; Return the value the children have computed.
            ,flag)))
```

Figure 1: Code for eband

3.2.1 Sockets on the ACL side.

First we need a function that will create a socket and connect the socket to the server socket. For simplicity I'm assuming that the server is going to be running on pollux.cse.Buffalo.EDU in this example.

```
(defun OpenConnection ()
   (let ((port '())
      (machine "pollux.cse.Buffalo.EDU"))
      (pprint "What port is the server being run on?> ")
      (setf port (read))
      ;;now actually make the socket, connecting at the same time
      (socket:make-socket :remote-host machine :remote-port port)) )
```

Now I'll define the function that will run as a separate process and listen to the socket for input. This method will read a string from a socket a line at a time (in case more than one line came in before the process got its next time slice). If the server is in C++, the strings will be null terminated and so the null character will have to be removed. If there are no nulls this code will also work.

```
   (defun ProcessInput (example-socket)
        (let ((InString "")
              (socket-output-lock (mp:make-process-lock)))
            ;;while we don't read a hangup, loop and play with the input
            (loop do
                ;;get the next input sentance - trim off nulls
                ;;from the beginning and the end (reading to the eol
                ;;will leave behind a null)
                (mp:wait-for-input-available example-socket)
                (setf InString (string-trim '(#\null)(read-line example-socket)))
                ;;uppercase the input and send it back to the server by writing
                ;;to the same socket we just read from
                (write-line  (string-upcase InString) example-socket)
                (force-output example-socket)
             until (equalp InString "Quit")
           )
         )
         (close example-socket)
    )
```

Now in the code of the program, we need to setup the socket and spawn off the new process that will listen to the socket. It is not necessary to spawn off a new process in this example since we will only listen for input on one stream, but it is a good idea to do so if you will listen on more than one stream since that allows us to wait for available input for each one and allow the processes to sleep if there is no input.

```
   .
   .
   (let ((example-socket (OpenConnection)))
     (mp:process-run-function "Sockets Example" #'ProcessInput))
   .
   .
```

The above code is sufficient to run a simple connection between a Lisp program and a program running in another language. Note that the Lisp program is both reading from and writing to the same stream. The sockets are of course two way and take care of making sure that the right program gets the right data. An example of this program running connected to the Java program shown in the next section will be found later in this paper.

3.2.2 An example Java server.

Java provides classes which make it easy to build a simple server. The networking infrastructure is included in the Java API libraries. The example below is a simple example of a Java-based server process to talk with the Lisp-based example given above. There is one problem with this example. The read should really be reading in a full line. <I'll correct this soon>. All code in Java is part of a class. This class is really a simple program unto itself. First before we define the class though, I would like to import the symbols for three of the Java API classes that will be used in this example so that I can simply refer to them by the class name without the package name.

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.OutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
 //Next we begin the class definition, setting up the instance variables for this object.
public class ServerTest {
  private ServerSocket exampleServer;
  private Socket exampleConnection;
  private OutputStream exampleWrite;
  private BufferedReader consoleInput;
  private BufferedReader clientInput;
  //Next we define the constructor for our class. Since it is such a simple example, we just
  //initialize all of the instance member variables here in the constructor.
  //Note that the constructor declares that it might throw an exception of type
  //java.io.IOException since some of the methods being invoked might
  //throw the exception and this constructor is not catching any exceptions.
  public ServerTest () throws java.io.IOException{
    //creating a server socket with a port of zero tells the server
    //to use any free port since on a multi user machine we are not
    //guaranteed that any port is necessarily free to use.
    exampleServer = new ServerSocket(0);
    System.out.println("The server is listening on port " +
       exampleServer.getLocalPort() + "\n");
    //program will wait here until the Lisp client connects to it
    exampleConnection = exampleServer.accept();
    exampleWrite = exampleConnection.getOutputStream();
    //create a buffered reader for standard in so that we can do a
    //readline in the next method
    BuffInput = new BufferedReader( new InputStreamReader (System.in));
    //now have a buffered reader for the connection to the Lisp program
    clientInput = new BufferedReader( new InputStreamReader
                                                         (exampleConnection.getInputStream()));
  }
  //This method reads loops reading strings from standard in and writes them to the stream
  //associated with the socket.
   public void ServerLoop() throws java.io.IOException{
     String line = null;
     String returnLine = null;
     //loop forever
      while (true) {
          //blocks till there is input available
          line =  consoleInput.readLine();
          if (line != null) {
```

```
            //send the information to the client example; add a newline and
            //then convert the string into a byte array to send it.
            exampleWrite.write((line + "\n").getBytes());
        }// end of if (line != null)
         returnLine = clientInput.readLine();
         if(returnLine != null)
             System.out.println(returnLine);
    } // end of while (true)
  }
//Finally we have the main method for the program;
//main methods are static with a set signature.
public static void main (String[] args) throws java.io.IOException {
    ServerTest example = new ServerTest ();
    example.ServerLoop();
  } // end of main ()
}//close the entire ServerTest class
```

3.2.3 Sample output from running the Java and Lisp programs

When we run these programs, we'll first run the Java program since its network connection is acting as the server waiting for a client connection. Then we will run the Lisp connection. So the output might look like the following, however please note that I broke the commandline into two lines so it would fit on a page, normally this should be on a single line. First we run the Java program:

```
java -classpath /projects/jsantore/johns-papers/SNeRG-Tech/java-test/:
/projects/jsantore/johns-papers/SNeRG-Tech ServerTest
The server is listening on port 36153
```

Now we need to start the lisp connection, so for simplicity I ran the two lines of code I needed from an ACL prompt after loading the program components listed in the Lisp section.

```
cl-user(41): (setf example-socket (OpenConnection))
"What port is the server being run on?> "36153
#<multivalent stream socket connected from pollux.cse.buffalo.edu/36170
to pollux.cse.buffalo.edu/36153 @ #x4fb8262>
cl-user(42): (mp:process-run-function "Sockets Example" #'ProcessInput)
#<multiprocessing:process Sockets Example @ #x4fbaa12>
```

Once this has been done the socket connection has been setup. We can just type input into the console window that the Java program was run on. The Java program will read that input and send it to the lisp program, which upcases it and sends this new string back to the Java program, which then prints it to the console. Note that these two functions could be reversed so that the lisp program reads and the Java program upcases the strings.

```
This demo shows a Java program talking to a Lisp program.
THIS DEMO SHOWS A JAVA PROGRAM TALKING TO A LISP PROGRAM.
They use socket connections to communicate.
THEY USE SOCKET CONNECTIONS TO COMMUNICATE.
The java program reads what I type from the console
THE JAVA PROGRAM READS WHAT I TYPE FROM THE CONSOLE
And then sends it to the lisp program which upcases it
AND THEN SENDS IT TO THE LISP PROGRAM WHICH UPCASES IT
and sends the upcased version of the text back.
AND SENDS THE UPCASED VERSION OF THE TEXT BACK.
quit
QUIT
```

3.2.4 An example server using BSD Sockets

The following is an example program written in C that uses the original socket methods without any wrappers. I used the tutorials [3, 4] to help me write this. It does something similar to the Java server above, but for the sake of space, it is slightly simpler in that it only sends one string to the Lisp program for upcasing. As you can see, since you are working with at a lower level, this program is much longer than its Java counterparts.

```
#include <errno.h> /* obligatory includes */
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <netdb.h>

void ServerConn(){
  int retval,
  new_sd; int sock; //socket 'file' descriptors
  int adrlen; /* sockaddr length */
  struct sockaddr_in myname; /* Internet socket name */
  struct sockaddr_in* nptr; /* ptr to get port number */
  struct sockaddr addr; /* generic socket name */
  char dataBuf[256]; //max 256 bytes for this simple example
  if (( sock = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
    printf("network server socket failure %d\n", errno);
    perror("network server");
    exit(1); }

  myname.sin_family = AF_INET; /* Internet address */
  myname.sin_port = 0; /* System will assign port # */
  myname.sin_addr.s_addr = INADDR_ANY; /* "Wildcard" */

  /* Bind the Internet address to the Internet socket */
  //we are casting to sockaddr for SUN version of BSD socket functions
  if (bind(sock, (struct sockaddr *)&myname, sizeof(myname) ) < 0 ) {
    close(sock); /* defensive programming */
    printf("network server bind failure %d\n", errno);
    perror("network server"); exit(2); }

//start getting the port number of the server:
adrlen = sizeof(addr); /* need int for return value */
if ( (retval = getsockname(sock, &addr, &adrlen ) ) < 0 ) {
    printf("setwork server getsockname failure %u\n", errno);
    perror("network server");
    close (sock);
    exit(3); }

  //following the the tutorials, doing this sort of pointer
  //manipulation is supposed to make things easier
  //to report the port number
  nptr = (struct sockaddr_in *) &addr; /* port # */
  printf("\tnetwork server: server has port number: %d\n", ntohl(nptr->sin_port));
```

```
  if ( listen ( sock, 2 ) < 0 ) {
    printf("C network server listen failure %d\n", errno);
    perror("network server");
    close (sock);
    exit(4); }

  if ( ( new_sd = accept (sock, NULL, NULL ) ) < 0 ) {
    printf("C network server accept failure %d\n", errno);
    perror("network server");
    close (sock);
    exit(5); }
  //now we should have a completely setup connection lets do
  //something with it. Actually, as mentioned, we'll keep this simple

  //first clear the buffer
  memset(dataBuf, '\0', sizeof(char[256]));
  strcpy(dataBuf, "This is a simple test of the Server in C \n");
  write (new_sd, dataBuf, sizeof(dataBuf));
  //now read from the connection for the upcased version
  //first clear the buffer
  memset(dataBuf, '\0', sizeof(char[256]));
  if ((read (new_sd, dataBuf, sizeof(dataBuf))) < 0 ) {
    printf("C version of network server ");
    printf("socket read failure %d\n", errno);
    perror("network server");
    close(new_sd);
    exit(7); }
  printf("C version of the network server received");
  printf(" the message %s\n", dataBuf);
  //tell the Lisp part of the program that we are done
  strcpy(dataBuf, "Quit \n");
  write (new_sd, dataBuf, sizeof(dataBuf));
  //we're all done now close the socket
  close(new_sd); }
}

main (int argc, char *argv[]){
  ServerConn();
}
```

To compile and link this program you'll need to link in the socket libraries. I did the whole thing in one step on our CSE systems with the following line:

`gcc -o c-server -lsocket cserver.c`

So the program will be named c-server and the code listed above is in a file called cserver.c.

3.2.5 Getting the example code.

The code for the examples used in this technical report are available on the University at Buffalo CSE systems in the following locations: For Professor Shapiro's eband macro see /projects/shapiro/eband.cl. For the various socket examples see the /projects/jsantore/johns-papers/SNeRG-Tech/{c-test|java-test|lisp-test} directories. All of the files are publicly accessible.

References

[1] http://www.franz.com/support/documentation/6.2/doc/multiprocessing.htm#process-locks-1

[2] http://www.franz.com/support/documentation/6.2/doc/socket.htm

[3] http://world.std.com/~jimf/papers/sockets/sockets.html

[4] http://www.ecst.csuchico.edu/~chafey/prog/sockets/sinfo1.html