LiPS: a system for distributed processing on workstations

Ralf Roth
Thomas Setz
FB-14 Informatik
Universität des Saarlandes
D-6600 Saarbrücken
Germany

29. Juni 1993

Zusammenfassung

LiPS (Library for Parallel Systems) is a collection of C functions enabling a programmer to distribute applications with low communication granularity over a network of UNIX workstations. LiPS restricts its applications to the use of idle time. As the potential computing power arising from wasted time slices (idle time) of workstations often exceeds even the power of supercomputers, LiPS is a cheap alternative to solve computing intensive problems. Based on UNIX using socket communication primitives, LiPS was developed on a network of workstations running the UNIX Operating system. The present release (2.1) is used at the Department of Computer Science of the Universität des Saarlandes; with 150 machines, we reach a total computing power of approximately 2500 MIPS.

Inhaltsverzeichnis

1 Introduction **4**

2 LiPS: A comprehensive survey **4**

2.1 The LiPS service 5

2.2 Techniques used to distribute LiPS applications 6

2.3 Monitoring of a LiPS application 8

3 Installation of LiPS **8**

3.1 What is needed in order to work with LiPS 9

3.2 Installing the network service 10

4 Monitoring of LiPS Applications **11**

5 Programmer's Guide **11**

5.1 General explanations 11

5.2 Setting up your environment: the configuration file 11

5.3 System variables 13

5.4 User functions 13

5.4.1 Initializing and stopping LiPS applications 14

5.4.2 Starting clients 15

5.4.3 Standard communication 15

5.4.4 Generative communication 17

5.4.5 Killing clients 21

5.4.6 Error handling 21

6 Examples **22**

6.1 Message Communication 22

6.1.1 The Master program 22

6.1.2 The Client program 22

6.2 Tuplespace Communication 23

6.2.1 The Master program 23

6.2.2 The Client program 24

7 Future Work **25**

1 Introduction

In the last decade a change of the common computing environments from large mainframes to local area networks of powerful workstations and personal computers took place. With the increasing amount of CPU power, more and more people thought about how to use the idle time, which is a result of this progress. Nowadays it is difficult to justify the waste of this computing power.

In order to recycle the available computing power we need a system which supports distributed processing in this environment. Therefore, we decided to design LiPS, a library of functions which allows a programmer to distribute his applications on a network of workstations without any knowledge in network programming.

There is a variety of systems which allow the distribution of applications. For example distributed operating systems would do this work in a transparent manner. If we try to recycle available idle time on machines not owned by us, we cannot able to put our own operating system on these machines. The operating system running on these machines has to be the interface between our library functions and the deeper layers. Therefore, LiPS is implemented on top of the operating system (BSD UNIX 4.2). Similar concepts are implemented in *Linda* [Bjo91] (Piranha project), *Condor* [LLM87], and *PVM* [BDG+91]. LiPS mainly differs from these systems in the interprocess communication mechanisms provided, including generative communication (Linda), and the restriction to the use of idle time.

In the second section, we explain the methods and concepts LiPS is based on. Section three deals with the installation of the system. Possibilities for monitoring LiPS applications are described in section four and the last section gives some examples.

2 LiPS: A comprehensive survey

A typical environment where LiPS could be applied consists of at least one network of UNIX workstations. In most cases several machines belong together by sharing one or more filesystems via NFS (Network File System). Such a group is called a "cluster".

The aim of our system is to use as much available computing power as possible without disturbing other users. Therefore its work is restricted to the use of idle time. We distinguish between several definitions of the term "idle time". In order to make LiPS available on many machines, it is implemented completely outside the UNIX kernel; no super user access is needed in order to install and work with the system. LiPS communication mechanisms are directly based on sockets (BSD UNIX 4.2). The current release has been implemented and tested on Sun Sparcstations running SunOS 4.1.1.

We divide the description of LiPS into three items:

- a service (`in.lipsd`) running on each workstation establishing access to those machines.
- methods enabling the user to distribute his program over the network.
- monitoring of applications.

2.1 The LiPS service

There are different possibilities to achieve remote execution in UNIX, e.g. rexec service, remote procedure call, remote shell. In former releases, we tried to use each of these possibilities. We noticed that those services often did not work as expected. As most of the owners of workstations do their job in a specialized area, the installation and configuration of their system can be trusted only for these "special purposes" because installation and configuration are only tested in this environment. In order to be independent from these disadvantages of a heterogeneous environment, we decided to implement our own service which provides us with access to all machines. This service, called `in.lipsd`, has to be installed on every workstation. No super user permission is needed in order to do this. `in.lipsd` is the basis for runtime distribution and monitoring of applications.

The combination of machines in our computing environment changes continuously because machines crash or are halted in order to do some system administration work. A service operating in such an environment should be aware of this fact.

When we designed our service we had three main goals in mind:

- security against unauthorized use,
- minimizing maintenance necessity,
- reliability of the distributed service.

Within our service, every message is marked by an authorization pattern. Messages having a wrong authorization pattern will be ignored.

In order to minimize maintenance necessities, the network reconfigures itself every time a member of the network is booted. We use `cron` and `at` jobs to implement this feature. Once installed, there is only little amount of LiPS system administration work.

One main task of our service is to make the addresses of the members of the network available in a reliable way. We handle this problem by applying two main concepts:

1. A ring of so-called *Fixserver* machines from which, we hope, at least one machine is running builds the basis of our net.

2. Every time a machine, *Fixserver* or not, is (re)started, it introduces itself to all the other members of the system with the help of one *Fixserver*.

The main advantage of more *Fixservers* is the reliability of the system in case of failure of a machine a *Fixserver* resides on. The consistency of the system data is guaranteed by protocols between the *Fixservers*. The CPU time the `in.lipsd` processes need is negligible simply because there is only little need for communication. Figure 1 describes the introduction of a new member in the network. As a result, every machine knows an address to reach every other machine.

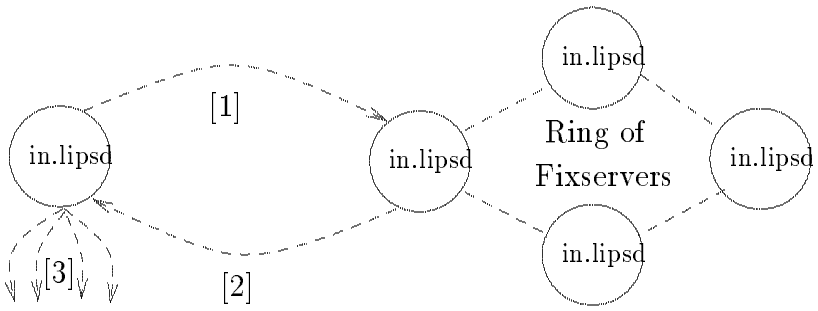

Figure 1:

Once started via `cron`, `in.lipsd` introduces itself to the ring of *Fixservers* [1] and gets the addresses of the other members of the network [2]. Now it is able to introduce itself to the other nodes [3].

2.2 Techniques used to distribute LIPS applications

There are several user functions to start, stop and coordinate distributed execution of our programs. A detailed description of these functions is given in section five of this report. In this section we explain the method distributed processes use to communicate. What every LIPS process does first is to call the function `initlips()`. This function creates a new (local) process called *daemon* which handles all communication with the help of local and remote `in.lipsds`. A user process and its *daemon* process communicate via a socketpair connection.

Unlike other systems for distributed implementations, we do not use RCP or RSH *daemons*.

There is a linear dependency between the CPU time needed by the *daemon* and the amount of communication. As there is no need for much communication in our applications, the runtime of the *daemon* is negligible.

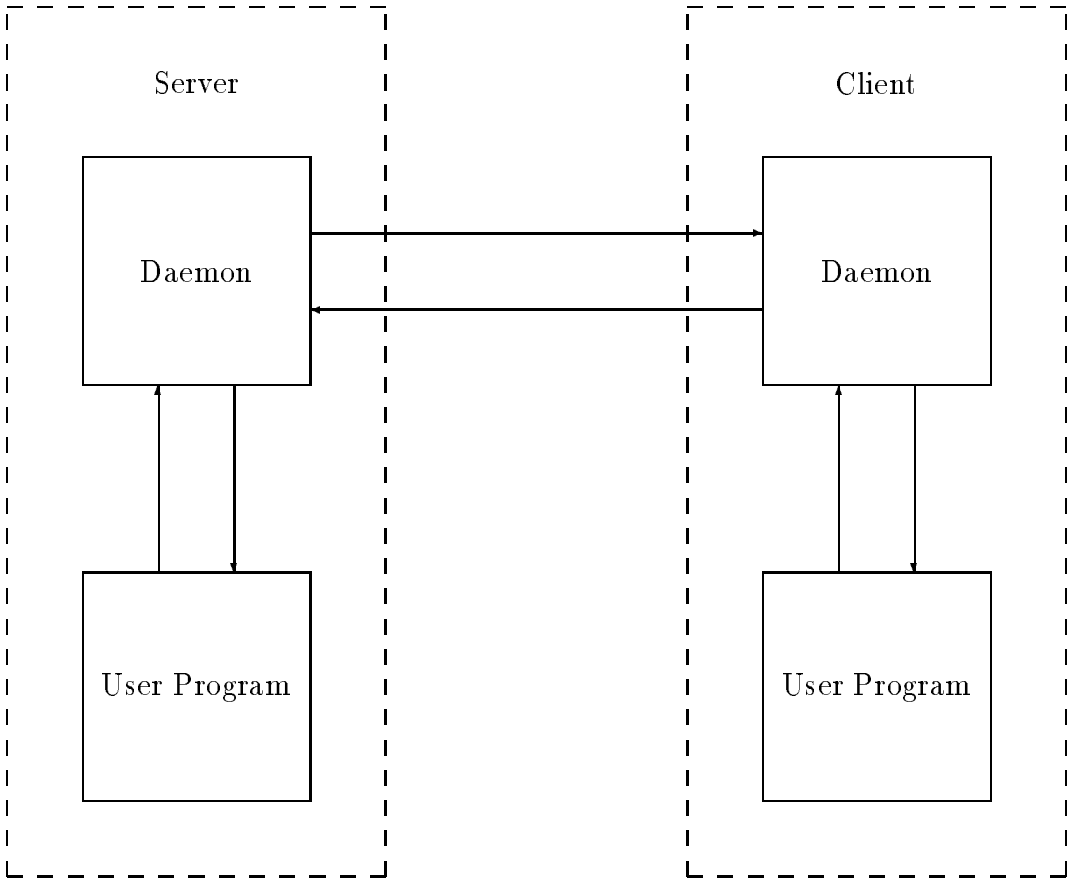

Figure 2:

Once started, a LIPS process forks a child process which handles all communication on the network. Internal communication between user process and daemon is done via a socketpair connection.

The whole network communication is managed by the *daemon*. The user program does not directly participate in the network communication. In particular, the tasks of the controlling *daemon* are the following:

- Waiting for communication requests from its user program or another *daemon*.
- In case of an external communication request; establishing a temporary socket-stream connection to the requesting *daemon*.
- Processing incoming messages from other *daemons* and the user program.
- Buffering messages .
- Checking if the user process has permission to compute at this time.

The user (owner) of a machine can grant the following permissions to LIPS:

- permission for a whole day,
- permission for a certain time interval during the night,
- permission when less than a specified number of users are logged in or all present users are idle on input.

These permissions are checked (current version) every 30 seconds. If the permission has changed since the last check, the `daemon` stops (or wakes up) the user program by sending a signal. Our processes remain in the run queue needing memory at least in the swap area. This might disturb the owner of the workstation if his swap area is small and the LiPS processes are large, a problem which will be solved in the next release.

2.3 Monitoring of a LiPS application

One main problem in a distributed environment is obtaining information about the progress of an executing program. As there is no fault tolerance in our system, it is important to get the state of each process of the application. The LiPS monitor makes such information available to the user with the help of the `in.lipsd` of each machine. Other functions support obtaining further information from each workstation in the network. A detailed description of these functions will be given in section 4.

3 Installation of LiPS

The LiPS package for Sun Sparcstation is delivered as is without express or implied warranty.

THE UNIVERSITÄT DES SAARLANDES AND THE LiPS DESIGN TEAM DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANDIABILITY AND SUITABILITY. IN NO EVENT SHALL THE UNIVERSITÄT DES SAARLANDES AND THE LiPS DESIGN TEAM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM THE LOSS OF DATA OR PROFITS, OR FROM ANY OTHER ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Permission to use, copy, modify and/or distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the copyright notice above appears in all copies, that both the copyright notice and the permission notice appear in supporting documentation, and that the names of the Universität

des Saarlandes and the LiPS Team will not be used in any advertisment or publicity pertaining to the distribution of the software without specific, written prior permission.

The LiPS package consists of:

- `liblips.a` : object library of user functions
- `lips.h` : LiPS header file
- `l_errno.h`: header files for error handling
- `_l_errno.h`: header files for error handling
- `in.lipsd` : binary of LiPS service
- `tron` : binary for starting and controlling `in.lipsd`
- `ptron` : bourne shell script for calling `tron` via `at` job
- `limon` : binary of LiPS monitor
- `lips.conf` : example of *Fixservers* file
- `CLIENT_CONF` : example of client configuration file
- `master.c` : example of LiPS master
- `client.c` : example of LiPS client
- `Makefile` : example Makefile
- `lips.dvi` : this documentation
- `installation.hints` : some installation hints

3.1 What is needed in order to work with LiPS

The files mentioned above are necessary to install LiPS.

A user account (we suggest the username `lips`) has to be available on each machine which will participate in a LiPS application.

Once in each cluster, a directory `bin` and a directory for each LiPS user has to be created in the home directory of the LiPS account. The LiPS user directory name is identical to the user's login name on the master machine, the workstation where the LiPS application is initiated.

Now the *Fixserver* configuration file, `lips.conf`, should be edited. Up to 5 *Fixservers* may be specified there. (Only those machines which should be running permanently are selected as *Fixservers*). Afterwards, the files `in.lipsd`, `tron`, `ptron` and `lips.conf` should be copied into the directory `$HOME/bin` once in each cluster.

The following steps are necessary only for those machines where LIPS programs are compiled and linked:

- The LIPS library `liblips.a` is copied into a library directory. In order to update the symbol table of the library, run `ranlib`.
- The LIPS header file is copied into an include directory.

The monitor program, `limon`, has to be available on machines where the distributed applications will be monitored.

3.2 Installing the network service

The processes establishing access to our network should run permanently. Therefore, they are frequently checked, e.g. every hour. The tools used are `cron` or `at` jobs calling `tron`. (See your UNIX Manual for further explanations on `cron` and `at`.) `tron` checks whether `in.lipsd` is running or not and starts it, if necessary. This leads to an automatic reconfiguration of our network every time a machine is booted and spares the user from administration work on the system.

The first thing we do is to establish the *Fixserver(s)* by calling `tron` on each of the machines specified in `lips.conf`. Then we establish a frequent `tron` call on every machine with the help of `cron` or `at`. The `crontab entry` has the following format:

```
0 * * * * $HOME/bin/tron > /dev/null 2>&1
```

If there is no access to `cron` jobs on a machine, you can use `at` in the following way:

```
at now + 1h ptron.
```

In order to split the introduction of machines in time, we suggest not to start `tron` at the same time on each machine.

4 Monitoring of LiPS Applications

In order to obtain knowledge about the progress of an executing distributed program or about the LiPS network itself, you have to start `limon`. There are two different views of the system: administrator's view and user's view. The administrator's view (username `lips` recommended) allows you to restart the whole LiPS network, in addition to the normal user capabilities. Moreover, under the LiPS view all user processes are shown whereas the user's view is restricted to those processes started by the user. First thing `limon` does is to ask you for your username and password. After identification, you enter the `limon` command interpreter. The following commands are available (and can be displayed online by the `help` command):

help	: display this text
end	: stop program
print	: show available hosts
ping ip_addr	: get information about a special host
netping	: get information about all hosts in the LiPS network
mipse	: give total number of available mips
netplat	: restart the whole LiPS network
proc ip_addr	: show processes on host
netproc	: show all processes in LiPS network
kill ip_addr	: kill all processes on host
netkill	: kill all processes in the LiPS network

5 Programmer's Guide

5.1 General explanations

The intended use of our system follows the master-worker paradigm of parallel computing; a so-called master process initiates and coordinates the distributed execution of applications. It starts so-called worker- or client processes which compute the work and deliver their results to the master process. Therefore, the master's main task is to start clients, distribute jobs, collect results, and last, but not least, kill the clients at the end of an application. The master is started under the user's account like a normal program, whereas the client processes are running with the LiPS user ID.

5.2 Setting up your environment: the configuration file

In order to provide the data concerning participating machines, the user has to specify a configuration file. This file contains the different machine names, internet

addresses in dot notation, the LiPS username, computing permissions, and cluster numbers. Lines beginning with # are ignored (comments). The first line which is not masked by a # sign gives the absolute path of `$HOME/lips`, where `$HOME` is the home directory of the LiPS account on the machine where the master process will reside. The data has to be specified in the following format:

```
#
# Parameter 1: Internet Address ( name of the remote machine where the
#              programs are to be executed)
# Parameter 2: Name of client machine in extended internet notation
#              (e.g: crypt1.cs.uni-sb.de)
# Parameter 3: LIPS   Username ( user's name on the master machine)
# Parameter 4: Daytime (Time [in hours] a  process is allowed to work)
# Parameter 5: Days (Time [in days] a process is allowed to work.
#        The week starts with Sunday.)
# Parameter 6: Number of users (maximal) on this machine
#              (more users ==> no processing time)
# Paramater 7: Machine is member of this cluster of NFS Groups
#              (send binaries to each first working member of a cluster)
#*****************************************************************
#
#This is the path where the local lips data is present
/home/lips
# This is the first cluster (so called Sparc Parc)
134.96.232.2  crypt2.cs.uni-sb.de      thsetz  20 8 0000000 1  1
134.96.232.11 crypt11.cs.uni-sb.de     thsetz  20 8 0000000 1  1
134.96.232.12 crypt12.cs.uni-sb.de     thsetz  20 8 0000000 1  1
134.96.232.13 crypt13.cs.uni-sb.de     thsetz  20 8 0000000 1  1
#
# sol establishes its own cluster
134.96.252.20 sol.cs.uni-sb.de        thsetz -1 25 1111111  100 2
#
# Now the third cluster is defined by mpi2000's
#
139.19.20.2     mpii02001.ag2.mpi-sb.mpg.de thsetz   -1 25 0000000 1 4
139.19.20.3     mpii02002.ag2.mpi-sb.mpg.de thsetz   -1 25 0000000 1 4
139.19.20.4     mpii02003.ag2.mpi-sb.mpg.de thsetz   -1 25 0000000 1 4
#
# Gipsy's establish the next cluster
134.96.228.2  gipsy1.cs.uni-sb.de      thsetz 7 21 1000001 1  5
```

The first entry of a line specifies the internet address of a machine, the second it's internet name in extended dot notation. The next parameter is the name of the user working with LiPS. The two following numbers define the latest end time in the morning and the earliest begin time in the evening for our processes to run. If you

define `-1 25` here this condition is never true. The following binary string defines days of the week when LiPS is allowed to work. Every digit represents a day of the week where the week starts with Sunday; 1 gives permission for this day. The number following this string determines the maximum number of users logged in on such a machine which will prevent processes from working. Additionally, processes will work even if all logged in users are idle on input, but only if this number is set to `0`. In this case, we only look at the permissions for whole days and hours. In addition to all these features which guarantee that the owner of a workstation is affected not hardly in his work, all processes are scheduled with lowest priority. The last number in every line represents the cluster a machine belongs to. A cluster is defined as a set of machines sharing a filesystem.

Example:

```
134.96.228.2  gipsy1.cs.uni-sb.de      laszlo 7 21 1000001 3  5
134.96.228.3  gipsy2.cs.uni-sb.de      laszlo -1 25 1000001 1  5
134.96.228.4  gipsy3.cs.uni-sb.de      laszlo 6 20 1000001 3  5
```

The first line shown above defines a machine with internet address 134.96.228.2 and name gipsy1.cs.uni-sb.de. Edit `/etc/hosts`, ask `nslookup` or see your system administrator to get your own data. The account owner LiPS processes will work on is `laszlo`. On this machine, LiPS applications will work from 9 p.m. to 7 a.m.. Furthermore, in principle, gipsy1 is allowed to compute every Saturday and Sunday and if either less than 3 users are logged in or all of them are idle on input. The second line gives permission when nobody is logged in on this machine or all users are idle. The last line in this example defines a machine where LiPS will work on Saturday and Sunday (whole day) and between 8 p.m. and 6 a.m. during the rest of the week.

The distributed package contains an example of a configuration file from our site.

5.3 System variables

Machine data is provided in a user accessible global array named `machine`. `machine` is a structured variable and consists of `machine_name` and `cluster_number`. For a detailed description of the data type, you may consult the header file `lips.h`.

5.4 User functions

These functions enable the user to start, stop, and control his distributed program. The initial process of the application is called master. This master starts the other (distributed) processes, called clients. The distributed package contains two files `master.c` and `client.c` as examples.

5.4.1 Initializing and stopping LiPS applications

The function `initlips()`

```
int initlips (client_conf, sleepfunc, exitfunc)
char *client_conf;
void (*sleepfunc)(),(*exitfunc)();
```

In order to initialize our system, we provide the function `initlips()` which, among others, takes the name of the configuration file as a parameter. It should be the very first function called in the user program. The call of `initlips()` is necessary to set up some well defined conditions. Now the LiPS functions can be used.

On the master machine the daemon reads the configuration file and stores all the information in an array of type `MACHINE`. In a client program this parameter has to be `NULL`. After calling this function, there exist two processes named `lips_prgname` on that machine. The first process represents the user algorithm, the second holds the daemon.

The parameters of `initlips()` are:

`char *client_conf`	name of the configuration file
`void (*exitfunc)()`	pointer to a function called by `exitlips()`
`void (*sleepfunc)()`	pointer to a function called before a client is suspended by the system

The return value of `initlips` is 0, if no error occurs. If there are any problems, `initlips` returns -1. More information about error handling is given in `l_error`.

The function `exitlips()`

```
int exitlips()
```

This function has to be called by the master process in order to end the program. It closes the internal socketpair connection and terminates user and daemon processes. On a client, `exitlips()` is automatically called whenever a client is killed. The function does not have any parameters.

5.4.2 Starting clients

The function `rstart()`

```
int rstart(program, rmachine, logfile)
char *program, *rmachine, *logfile;
```

The function `rstart()` is used to start a process on a remote machine. Its use is restricted to the master process. The command `rstart()` tells the server daemon to copy the specified program to the given remote machine, if it is the first reachable machine in a cluster, and to start it. The program's binaries must reside on the master machine.

Standard output and standard error of the client are directed into a file specified by `logfile`. If `logfile` is `NULL`, all output is sent to `/dev/null`.

The parameters of `rstart()` are:

`char *program`	name of the program to start
`char *rmachine`	name of the client machine
`char *logfile`	name of the logfile (`NULL` allowed)

The function `rstart_all()`, which complements the function `rstart()`,starts all reachable clients listed in the client configuartion file with the given program.

```
int rstart_all(program, logfile)
char *program, *logfile;
```

The parameters of `rstart_all()` are:

`char *program`	name of the program to start
`char *logfile`	name of the logfile (`NULL` is allowed)

All `logfile` names are automatically extended by the machine's internet address to avoid overwriting files in a NFS environment.

5.4.3 Standard communication

In LiPS, processes can communicate in two different ways:

- sending and receiving messages,
- copying files.

The function `netwrite()`

```
int netwrite(rmachine, message)
char *rmachine, *message;
```

`netwrite()` sends a message, restricted to 1024 bytes, to a given machine. Only those workstations where the LiPS application is running can be reached.

The parameters of `netwrite()` are:

`char *rmachine`	name of the remote machine (master is `NULL`)
`char *message`	message to send

The function `netread()`

```
int netread(rmachine, message)
char *rmachine, *message;
```

`netread()` tries to read a message from the specified machine. If a message is pending, it is copied to `message`. If there is no error, the number of available messages from this machine is returned. If there are no queued messages, 0 is returned. As usual, an error is indicated by a return value of -1.

The parameters of `netread()` are:

`char *rmachine`	name of the remote machine (master is `NULL`)
`char *message`	memory for received message

The function `netread_any()`

```
int netread_any(rmachine, message)
char *rmachine, *message;
```

`netread_any()` works as `netread()` except that a message from any machine may be read. The name of the machine is copied into `rmachine` and the message is copied into `message`. The oldest message of the machine with the longest message queue is copied.

The parameters of `netread_any()` are:

`char *rmachine`	memory for the name of a machine
`char *message`	memory for received messages

The function `rcp()`

```
int rcp(rmachine, source, target)
char *rmachine, *source, *target;
```

`rcp()` copies a file named `source` to `target` on the remote machine. The `rcp()` function may be used only if the corresponding client is running. This function is not based on the UNIX remote copy function.

The parameters of `rcp()` are:

`char *rmachine`	name of the remote machine
`char *source`	name of the source file
`char *target`	name of the target file

5.4.4 Generative communication

This model of communication needs a short explanation. It is derived from Linda [ACG86] and introduces a global accessible associative shared memory model. There are several Linda implementations differing in syntax, functions provided, distribution schemes, and fault tolerance.

Linda is a programming framework of language independent operators. Since our system provides us with a run time solution for Linda, those operators are comparable to system calls. Those system calls allow cooperation between parallel processes by controlling access to a logical shared memory called tuple space. The tuple space is a set of tuples which are simple collections of data. The data type of a tuple is determined by the arity (formal or actual) and the data type of the fields the tuple consists of. The manipulation of this memory is only possible via Linda calls which are syntactically function calls having a variable number of parameters.

The concept of tuple space (The following description is taken from [BM89])

"In most parallel programming languages explicit relationships between parallel processes are introduced either through the direct exchange of messages between processes or through the introduction of shared monitor processes, which must be specified according to the application and whose services can be accessed sequentially by other processes.

In contrast, process cooperation in Linda is reduced to the concurrent access of a large shared data pool, thus relieving the programmer from the burden of having to consider all process interrelation explicitly. The parallel processes are decoupled in a very simple way, leaving only one external interface to the shared data pool.

...
The shared data pool in the Linda concept is called *Tuple Space* (TS). Its access unit is the tuple, a list of data elements similar to the parameter list of a procedure. Since size and type of a tuple is user-defined according to the problem's requirements, the number of potentially costly TS accesses is reduced to the necessary minimum. Reading access to tuples in TS is not based on physical addresses - in fact, the internal structure of TS is hidden from the user - but on their expected content described in so-called *templates*. This method is similar to the selection of entries from a data base. Each list element of a tuple or template is either an actual parameter, i.e. holding a value of given type, or a formal parameter, i.e. a placeholder for such a value. Tuples in TS are selected by a matching procedure, where a tuple and a template are defined to match, if they have the same structure (corresponding number, type and order of elements) and if their actual parameters in corresponding places have identical values.

Linda defines five operators which are added to a conventional procedural language (e.g. C, Fortran or Modula-2) and enable the processes specified in this language to access the Tuple Space. These operators are translated into kernel calls by the compiler and are executed by the run-time system. These are Linda's operators:

`OUT(u)`	generates a tuple u and inserts it into TS, this does not block the calling process.
`IN(e)`	selects a tuple u from TS which matches template e and removes it from TS. If no matching tuple exists the calling process is suspended until another process generates such a tuple.
`READ (e)`	same as `IN`, but the tuple remains in TS.
`INP(e)`	non-blocking `IN`, returns FALSE, if no matching tuple exists (predicate function).
`READP(e)`	non-blocking `READ` (predicate function).

The result of a tuple selection is the assignment of the actual values in the tuple u to the corresponding formal variables of the template e. The Linda operators, though very simple, are powerful aids for the communication and synchronization of processes. Access conflicts do not occur because tuples can not be updated in TS directly, but must always be removed from the TS and reinserted in their updated version. In the meantime they are not available and can therefore never be read in an inconsistent state by other processes."

The function `out()`

```
int out(destination, logic_name, type_descriptor, var_args)
```

```
char *rmachine, *logic_name, *typ_descriptor;
va_list *var_args;
```

`out()` puts a tuple into the local tuple space of `destination`. If `destination` is a NULL pointer, the tuple will be put into the local tuple space, i.e. tuple will be held by the local daemon.

Every tuple has a logical name which is used to speed up tuple matching.

The type descriptor and the variable list are the main parts of this call. The type descriptor describes the data type and arity of the variables defined in the variable list. The following data types are implemented:

`i`	integer
`s`	string
`l`	long
`f`	float
`d`	double
`c`	character.

`out()` does not allow formal parameters, it takes the values of the defined variables (variable list) and interprets them according to the types specified in the type descriptor. The data is put into `destination`, where it can be read with the help of `in()` (destructive read) or `rd()` (non-destructive read).

The function `in()`

```
int in(logic_name, typ_descriptor, var_args)
char *logic_name, *typ_descriptor;
var_list *var_args;
```

First the function `in()` looks for a match in the local part of the tuple space. If there is no local match, the search is expanded to the whole tuple space. `in()` is a blocking operation.

`in()` delivers a matching tuple (formerly created by `out()`) and binds the given values to those variables which are defined as formal in the type descriptor. Contrary to the `out()` call, there are two different kinds of type descriptors:

- formal (`I S L F D C`) and
- actual (`i s l f d c`).

Formal parameters describe those variables which will be overwritten by the new value. Actual parameters describe, like the logic name, the format of the matching tuple. A message put into tuple space via `out()` matches an `in()` call if:

1. the logical names are the same,
2. the number and data types of the fields correspond ,
3. the values of actual parameters are identical.

If there is a match, the message is taken out of tuple space and the formal variables are overwritten with the given values.

In order to find a match, `in()` first looks for a match in the local tuple space. If this request cannot be satisfied locally, we broadcast it to all the other nodes participating in the application.

In order to minimize communication overhead, tuples should be put into the local space of the machine that will most probably ask for it.

The function `rd()`

```
int rd(logic_name, typ_descriptor, var_args)
char *logic_name, *typ_descriptor;
var_list *var_args;
```

This function behaves like `in()`, but leaves a matching tuple in tuple space.

The function `inp()`

```
int inp(logic_name, typ_descriptor, var_args)
char *logic_name, *typ_descriptor;
var_list *var_args;
```

Contrary to `in()`, `inp()` is a non-blocking function. It returns a value of -1 if there is no `local` match. This is a difference from all other known Linda systems and it is based on the possibility to deliver tuples directly to a node. The broadcast is done similar to `in()`.

The function `rdp()`

```
int rdp(logic_nameP, typ_descriptor, var_args)
char *logic_name, *typ_descriptor;
var_arg *var_args;
```

This function is similar to `inp()`, except that the matching tuple remains in the tuple space.

5.4.5 Killing clients

The function `kill_client()`

```
int kill_client(rmachine)
char *rmachine;
```

`kill_client()` stops a user program as well as its daemon on a given remote machine.

There is also a function `kill_all_clients()` which stops all clients. This function has neither parameters nor a return value.

5.4.6 Error handling

The error handling in our system is implemented similar to UNIX. In case of success, a function returns 0, otherwise -1. If an error occurs, the global variable `l_errno` is set to the fault's value. The corresponding error message is made available by the function `l_error`. In order to use this function, the header file `l_errno.h` has to be included.

Examples are given in `master.c` and `client.c`.

The function `l_error()`

```
int l_error(string)
char *string;
```

Use this function like the UNIX function `perror()`. See your manual for further explanation.

6 Examples

In this section we give some examples of how to use LiPS-functions. The distributed program computes the Mandelbrot set. We divide it into two parts, a master and a client. The master process distributes the work to the clients and collects their results. The first example uses message communication, the other our implementation of Linda calls.

6.1 Message Communication

6.1.1 The Master program

```
#include <lips.h>

char    machine[64], message[1024];
int     pixel_num = 1000 * 1000;       /* we use 1000 * 1000 pixels */
int     index;
int     xoffset[4] = {0, 500, 500, 0};
int     yoffset[4] = {500, 500, 0, 0};

main()
{
    initlips("configfile", NULL, NULL);
    rstart_all("clientprg", "logfile", 1);
    netwrite(machine_name[0], "-2 0  0 2");
    netwrite(machine_name[1], " 0 2  0 2");
    netwrite(machine_name[2], " 0 2 -2 0");
    netwrite(machine_name[3], "-2 0 -2 0");
    while (pixel_num--) {
        while (netread_any(machine, message) == 0)
            sleep(1);
        sscanf(message, "%d %d %d", &x, &y, &c);
        x += xoffset[index = getindex(machine)];
        y += yoffset[index];
        "set pixel at location (x,y) with color c"
    }
    kill_all_clients();
    exit_lips();
}
```

6.1.2 The Client program

```
#include <lips.h>
```

```
char    message[1024];
double  left, right, bottom, top, y_step, x_step;
int x, y, escape;

main()
{
    initlips(NULL, NULL, NULL);            /* initialize LIPS     */
    while (netread(NULL, message) == 0)    /* wait for your range */
        sleep(1);
    sscanf(message, "%lf %lf %lf %lf", &left, &right, &bottom, &top);
    x_step = (right - left) / 500.0;
    y_step = (top - bottom) / 500.0;
    for (x = 0; x < 500; x++, left += x_step)
        for (y = 0; y < 500; y++, bottom += y_step) {
            escape = divergence_speed(left, bottom);
            sprintf(message, "%d %d %d", x, y, escape);
            netwrite(NULL, message);      /* now send back your computation */
            }
    for ( ; ; )                           /* wait to be killed */
        sleep(1);
}

int divergence_speed(x, y)
double x, y;
{
    /*  Test, whether the given point is element of the Mandelbrot set */
}
```

6.2 Tuplespace Communication

6.2.1 The Master program

```
#include <lips.h>


char  machine[64];
int pixel_num = 1000 * 1000;
int index;
int x,y,c;
int xoffset[4] = {0, 500, 500, 0};
int  yoffset[4] = {500, 500, 0, 0};


main()
{
initlips ("configfile", NULL, NULL);
```

```
rstart_all("clientprg", "logfile", 0);

/*
 * Tell the clients what their work is (area to compute in) and
 * deliver these values into the clients' local space.
 * Within the master process, the enumeration of machines,
 * (machine[i]), is defined by their order in the configuration
 * file.
 */

out(machine_name[0],"area","ffff",-2, 0,  0, 2);
out(machine_name[1],"area","ffff", 0, 2,  0, 2);
out(machine_name[2],"area","ffff", 0, 2, -2, 0);
out(machine_name[3],"area","ffff",-2, 0, -2, 0);

while (pixel_num --) /* as long as there are results */
{

/* get any pending result . If you want to get the number of
iterations at location (5,2), (x,y) should be set to this value
and the call to in() would look like in("result","iiI",x,y,&c).
*/
in("result","III",&x,&y,&c)

x += xoffset[index = getindex(machine)];
y += yoffset[index];

/*
 * print pixel at location (x,y) with colour c
 */

}

/* kill your clients */
kill_all_clients();
exitlips();
}
```

6.2.2 The Client program

```
#include <lips.h>

double  left, right, bottom, top, y_step, x_step;
int x, y, escape;


main()
```

```
{
     initlips (NULL, NULL, NULL);

/*
 * Read the values of the area which should be computed
 */

in("area","FFFF",&left, &right, &bottom, &top);

x_step = (right - left) / 500.0;
y_step = (top - bottom) / 500.0;


for ( x = 0; x < 500; x++,left += x_step)
for (y = 0; y < 500; y++, bottom += y_step)
{
escape = divergence_speed(left, bottom);

/*
 * Within the client, machine[0] holds the master and
 * machine[1] holds the parameters of the client itself.
 * The following elements hold the other addresses of the participating
 * machines.
 */

out(machine[0].machine_name,i"result","iii",x,y,escape);
}

for(;;)
sleep(1);

}

int divergence_speed(x, y)
double x,y;
{
/* This function computes the number of iterations which are the
 * indicator for membership to the mandelbrot set
 */
}
```

7 Future Work

As we found out, using the idle time of workstations is a good solution for achieving much computing power. The choice of UNIX workstations and sockets as underlying communication tools makes it possible to develop large amounts of (cheap) computing

power. We believe that systems like LiPS will find their way from "scientific computing" to real life applications, but we also know that there are still problems in using this power. They can be divided into two areas.

First is the reliability of this power. As machines often crash or are halted in order to do system administration work, they lose data which might be necessary to terminate the application.

Second, such a complex system like ours should be easily installed and used. Therefore, graphical user interfaces at the system administration and user level will be designed.

Literatur

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *IEEE Computer*, August 1986.

[BDG+91] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and S. Vaidy. A users's guide to PVM. Technical Report ORNTL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, Tennessee, July 1991.

[Bjo91] R. Bjornson et al. Experience with Linda. Technical Report YALEAU/DCS/TR-866, Yale University Department of Computer Science, August 1991.

[BM89] L. Borrmann and Herdieckerhoff M. Linda für die Parallele Programmierung Konzepte, Implementierung und Leistung. Technical Report BeG 008/89, Siemens (München) int. Bericht, June 1989.

[LLM87] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor - Hunter of idle Workstations. Technical Report 730, University of Wisconsin - Dept. of Computer Science, Madison, December 1987.