A Distributed LISP-STAT Environment

David C. De Roure and Danius T. Michaelides

Department of Electronics and Computer Science
& Department of Social Statistics
University of Southampton, Southampton SO17 1BJ, UK

1 Motivation

With the advent of networking and high-powered workstations, and the rise of end-user computing alongside the traditional centralised computing model, the *heterogeneous network* is emerging as the most significant platform for many computing activities. A heterogeneous network consists of a number of resources (e.g. workstations, fileservers, database engines and computation nodes) interconnected by a fast data network; open systems standards facilitate interoperability in this (possibly multivendor) environment.

This platform offers significant benefits for statistical computing applications. Not only can programs access repositories of online data as required, they can also access other processors in order to exploit specialised features or just to harness idle resources. Hence computationally intensive tasks can be accelerated by farming out subtasks to multiple processors. Further, this platform supports groups of users and hence there is scope for the development of applications which facilitate collaborative working, such as a team of researchers working on a common dataset.

2 LISP-STAT

LISP-STAT [5] is a statistical computing environment based on the Lisp language; it is available on a variety of processors and operating systems. Lisp is a *dynamic language* and as such lends itself to distributed applications—its flexibility, interactive nature and reflective ('code as data') properties make it an effective means of supporting multiple cooperative processes on a heterogeneous network. For these reasons, LISP-STAT is an ideal basis for the development of a statistical computing environment for heterogeneous networks.

LISP-STAT provides dynamic, interactive graphics. Under the X-windows system, LISP-STAT processes running on one workstation can use another for their display and user interaction. These facilities alone are sufficient to allow one user to run LISP-STAT processes on multiple computers, each having its display set to the user's workstation. However, to harness the full potential of such an environment, mechanisms for communication between those processes must be provided—and these mechanisms need to be supported within the LISP-STAT language so that they are available to a user writing LISP-STAT programs.

3 Distributed Lisp

To establish an environment for distributed statistical computing, it is essential to specify standard interfaces between the components of the system. This will enable the construction of applications from a variety of distributed components complying with the interface conventions.

Our initial experiments have been with LISP-STAT processes communicating with other LISP-STAT processes, but we are now extending this so that they can communicate with programs written in other Lisp dialects and other languages (such as C). In the future, the components of the system may be other application programs which we can harness to comply with the interface. We envisage that LISP-STAT itself will be a front-end to such a hybrid system, and that the Lisp language itself will continue to provide the 'glue' and programmability of our system.

The utility of a *message-passing model* for distributed Lisp applications has been demonstrated [2]; this is one of the many models which we can build using primitive building blocks for concurrency and synchronisation [1]. Tierney [6] has proposed to adopt a broadcasting mechanism in Lisp (called *announcements*) as a future development of LISP-STAT. Like the message-passing models, we are adopting techniques which provide a concept of process location [3], upon which higher-level facilities (such as announcements) can be constructed. It is this which distinguishes our approach from the traditional vision of parallelism solely for speedup—we are able to migrate processes towards data.

4 Communications model

We have investigated a number of different communications models for use in our prototype system. To promote interoperability of components, we anticipate that more than one of these would be supported. The following models have been examined in detail, by building simple implementations and comparing them; we are still evaluating CORBA (the *Common Object Request Broker Architecture*), and the emerging MPI (*Message Passing Interface*) standard.

- **RPC** or Remote Procedure Call, offers a strict client-server model, i.e. a server publishes the availability of a service and remote clients can then make procedure calls to the server. RPC follows a well defined standard, and has the advantage that it is machine independent and provides a consistent programming interface which, at its highest level, resembles the familiar procedure call model. The RPC model is very strict (the client supplies some data, the server processes it and sends a reply, the transaction terminates) and although it can be relaxed, it is not designed to work with processes which are to behave as both clients and servers.

- **Unix sockets** provide a mechanism for two processes to communicate freely in a bi-directional manner. It requires that a process first set up a known connection point, then act like a server, accepting connections from client processes. Once communications are established, two processes can communicate freely; the connection may be transitory or it may persist for a number of exchanges. Sockets provide independence from the details of particular networking technologies.

- **TLI** or Transport Level Interface, provides a similar model to sockets, but it complies with OSI standards. The programming interface is more sophisticated than sockets and can lead to better engineered solutions. Although TLI is set to replace sockets on UNIX machines, the socket interface is only just becoming widely available on PCs.

- **PVM** or Parallel Virtual Machine aims to harness a collection of heterogeneous machine into a large distributed-memory computer. It provides a message handling system, as well as message parsing, process spawning and configuration library calls. It has emerged as a standard within the parallel processing community.

- **ToolTalk** aims to provide an interface to allow tools (such as editors, compilers, mailers etc) to be aware of what the user is doing in the environment. Tooltalk provides an announcement-like mechanism, as well as point to point communications.

Initially we will build our message passing model on sockets for our prototype; by careful design, the model will migrate to some of the other standard interfaces. The socket model provides a lightweight set of communication primitives, hence is an easy interface to implement, and its simplicity means that we can investigate and implement other communication strategies on top of this interface. Implementations of the socket library exist for both Macs and PCs, and a number of internet tools are based on this interface.

5 Low-level Lisp interface

The model used is that a process sets itself up to listen for connections at a known location, i.e. on a known machine, and a known 'subaddress' called a *port*. In this situation, it is acting like a server. There are a number of routines that set up such a connection. Firstly `socket` creates a new socket, returning a handle. This socket is then bound to a port using `bind`. Finally, the maximum number of queued connection requests is set using `listen`.

At this point the server is ready for clients to call it, which they do by making a socket and then using the `connect` function. On the server, the `accept` function waits until a client makes a call to the specific port, establishes the communication channel, and returns a handle onto the local end of the communication (a new socket). Another client process can then request a connection from this server, by again calling `connect` with the the machine location and the port number. Once the link has been established, the communication can be two way and can persist as long as necessary, i.e. not in the strict client-server model imposed by RPC.

There are three basic data transfer primitives, corresponding to the three basic types integer, float and string. All other types can be transferred by using these functions or combinations of functions and standard conversions.

Although a LISP-STAT process only has a single thread, it can have more than one connection talking to it. Care has to be taken, since if a read activity is initiated, it will not terminate until the data arrives (the read is said to 'block'). A mechanism, based on the UNIX `select` library call, allows you to identify which connections have data ready to be read on them.

6 Encoding data

Data in Lisp takes the form of symbolic expressions (*s-expressions*). In order to transfer data between Lisp processes, the data has to be serialised and transferred over the communication channel. Given the basic primitives to send an integer, string or floating point number, we can build up our protocol to handle any type of Lisp data. For each of the basic Lisp types, we assign an integer code. To send the data we first send the integer code so that the receiver knows what to expect. What follows is dependent on the type of the data. For example to send an integer, all that is sent is the code for an integer, followed by the integer itself.

For non-trivial pieces of data such as `cons`, recursive calls to a sending function are made. So for example a `cons`-pair (the basic unit of list structure in Lisp) consists of sending the code for `cons`-pair followed by sending the `car` expression (head of the list), then the `cdr` expression (tail of the list). This is illustrated in this fragment of code:

```
(defun send-sexp (socket obj)
  (case (class-of object)
    (integer (send-int socket int-code) (send-int socket obj))
    (cons    (send-int socket cons-code) (send-sexp socket (car obj))
                (send-sexp socket (cdr obj)))
    ...))
```

This strategy works fine, in general, but for cyclic data structures the recursive nature causes an infinite loop. This can be avoided by both the transmitting and receiving processes maintaining a list of nodes sent previously. When a node that has been sent before needs to be sent again, a reference to the earlier transmission is sent instead. The cache on the transmission end is based on a hash table to permit fast lookup based on the node itself; on the receive end it is a vector whose elements can be referenced in constant time.

In general, cyclic user data structures are not common, but there are two important exceptions in an object-oriented Lisp interpreter: functions and objects. Typically when we consider transporting functions, we need to consider the environment that the code was defined in (i.e. the values of variables external to a function but in scope when the function was defined), and this may be cyclic; similarly, implementations of objects and classes can contain cyclic structures. The transmission of objects is a similar problem to saving an object to a file, as pointed out by Tierney [5].

7 Example

Having implemented the communications primitives, we were in a position to investigate some form of distribution. As a useful demonstrator and tutorial application, we chose to implement a farm model. A master process is given a list of work that needs to be done. It then waits for requests from *workers*, and delivers each worker an item from the list. The worker then performs the task that is required using the supplied data, and then responds with the result. The master process then collects the results into a list and returns.

This is a relatively straightforward method of distributing work, and for certain applications give useful speedup. The communications overhead is typically very low, since the problem is split up into independent tasks, requiring minimal data transfer at the beginning and end. Course grained computations like this, with little synchronisation, are ideal for distribution.

The example used was the estimation of π by picking random points in a unit square, and testing if the points lie within an enclosed quadrant of a circle of unit radius. The square can be divided up into smaller regions and each region assigned to a process. Hence the only communications are sending the coordinates of the range and returning a ratio of points in the circle to points out of the circle. Another method is to assign a number of iterations to each processor, letting each processor pick random points across the whole square. This example is well-suited to distribution.

It is evident that this system cannot perform well for distributed applications where there is a substantial amount of communication between separate processes. As recognised previously, applications that perform well using this system are those which involve a high degree of computation that can be done in parallel. There are many statistical applications that match this model, typically involving intensive simulation. For example, estimating p-values using Monte Carlo methods, Gibbs sampling using multiple Markov-chains, computer-intensive resampling methods, and jackknife methods.

8 Conclusion and future work

We have established one interface for communicating LISP-STAT processes, based on a message-passing communication model, and demonstrated the use of this to accelerate a simple coarse-grained computation. We are currently generalising this interface, initially to support compiled Lisp systems.

Our prototype implementation will be used to provide support for two statistical applications: the first is the computationally-intensive Monte Carlo approach to the analysis of large sparse social mobility tables proposed by Smith and McDonald [4]; the second is a modified approach to a Markov Chain Monte Carlo (Gibbs Sampler) technique for the analysis of contingency tables. We believe that both applications will gain significant benefit from the distributed LISP-STAT environment.

References

[1] Neil Berrington, Peter A. Broadbery, David C. De Roure, and Julian A. Padget. EuLisp Threads: A Concurrency Toolbox. *Lisp and Symbolic Computation*, 6(1/2):177–200, 1993.

[2] David C. De Roure. Experiences with Lisp and distributed systems. In *Proceedings of the High Performance and Parallel Computing in Lisp Workshop*, Twickenham, London, UK, November 1990.

[3] Christian Queinnec and David C. De Roure. Design of a concurrent and distributed language. In Robert H. Halstead, Jr and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications*, Lecture Notes in Computer Science, pages 234–259. Springer-Verlag, October 1992. LNCS 748.

[4] Peter W. F. Smith and John W. McDonald. Simulate and Reject Monte Carlo Exact Conditional Test for Quasi-independence. In *Proceedings of COMPSTAT 1994*.

[5] Luke Tierney. *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*. J. Wiley and Sons, New York, 1990.

[6] Luke Tierney. Announcements in LISP-STAT. In *Bulletin of the International Statistical Institute, Proceedings of the 49th Session*, pages 117–124, August 2–September 2 1993. Book 3.