SPECIAL ISSUE

7th SA COMPUTER RESEARCH SYMPOSIUM
The South African Computer Journal
An official publication of the South African Computer Society and the South African Institute of Computer Scientists

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
Email: dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof John Schochet
University of the Witwatersrand
Private Bag 3
WITS 2050
Email: 035ebrs@witsvma.wits.ac.za

Production Editor
Prof G de V Smit
Department of Computer Science
University of Cape Town
Rondebosch 7700
Email: gds@cs.uct.ac.za

Editorial Board

Professor Gerhard Barth
Director: German AI Research Institute

Professor Pieter Kritzinger
University of Cape Town

Professor Judy Bishop
University of Pretoria

Professor F H Lochovsky
University of Toronto

Professor Donald Cowan
University of Waterloo

Professor Stephen R Schach
Vanderbilt University

Professor Jürg Gutknecht
ETH, Zürich

Professor S H von Solms
Rand Afrikaanse Universiteit

Subscriptions

<table>
<thead>
<tr>
<th></th>
<th>Annual</th>
<th>Single copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Southern Africa</td>
<td>R45,00</td>
<td>R15,00</td>
</tr>
<tr>
<td>Elsewhere</td>
<td>$45,00</td>
<td>$15,00</td>
</tr>
</tbody>
</table>

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Computer Science Department, University of Pretoria, Pretoria 0002
Phone: (012) 430-2504  Fax: (012) 43-4654  email: dkourie@dos-lan.cs.up.ac.za
7th Southern African Computer Research Symposium

Karos Indaba Hotel, Johannesburg
1 July 1982

PROCEEDINGS

Guest Editor: Judy M Bishop

Organised by the SA Institute of Computer Scientists in association with the Computer Society of SA

Sponsored by Persetel and the FRD
When the first SA Computer Symposium was held at the CSIR in the early eighties, it was unique. There was no other forum at the time for the presentation of research in computer science. In the intervening decade, conferences, symposia and workshops have sprung up in response to demand, and now there are several successful ventures, some into their third or fourth iteration. Each of these addresses a specific topic - for example, hypermedia, expert systems, parallel processing or formal aspects of computing - and attracts a specialised audience, well versed in the subject and eager to learn more. For the main part, the proceedings are informal, and certainly not archival.

SACRS, though, is still unique, in that it deliberately covers a broad spectrum of research in computing, and in addition, seeks to provide a lasting record of the proceedings. To achieve the second aim, we negotiated with the SA Institute of Computer Scientists for the proceedings to form a special issue of the SA Computer Journal, and the copy you have in front of you is the result. The collaboration between the symposium committee and the journal's editorial board placed high standards on the refereeing and final presentation of the papers, to the symposium's benefit, while we were still able to maintain a fresh, audience-oriented approach to the selection of papers.

This is SACJ's first such special issue, and the largest issue (at 145 pages) to date. We hope that it is only the beginning of future such collaborations.

In all 29 papers were received, all were refereed twice, and 19 were chosen for presentation by the programme committee. All the papers were thoroughly revised by the authors on the basis of the referee's comments, and the committee's suggestions aimed at making the material more accessible to a broadly-based audience. Papers had to be new, and not to have been presented elsewhere, a requirement that is still unusual within the SA conference round.

A third goal of SACRS has been to invite keynote speakers, usually from overseas. This year, we are fortunate to present Dr Vinton Cerf, the father of the Internet and a world-renown expert on computer networks. Although his paper is not available for this special issue, it will appear later in SACJ. Through the good offices of Professor Chris Brink of UCT, we also have three other speakers from Germany, Canada and the US adding interest to the event, and two of their papers appear in this issue.

The programme committee originally devised a theme for the symposium - "Computing in the New South Africa". We received several queries as to the meaning of this theme, but unfortunately few papers that addressed it directly. One prospective author went as far as to enquire whether computer research would survive in the new South Africa. Another felt that his work was definitely not in the theme, as it was genuine, old world, basic, theoretical science! Nevertheless, there are two papers that consider one of South Africa's key issues, that of language. Others look at the success we have achieved in applying technology to mining, and the future of low-cost operating systems. In all, the mix of papers represents a balance between the theoretical and the practical, the past and the future, all firmly based in the computing of the present.

Organising the symposium has involved the hard work of several people, and I would like to thank in particular

- Derrick Kourie, my co-organiser, and the editor of SACJ for his invaluable advice and hard work throughout the planning and implementation stages;
- Riel Smit, the production editor, for attaining such a high standard in such a short time for so many papers;
- Gerrit Prinsloo and the staff at the CSSA for their efficient and quite delightfully unfussy organisation;
- Persetel for their very generous sponsorship of R25000, and Tim Schumann for taking a genuine interest in our events;
- the Foundation for Research Development for sponsoring Vint Cerf's visit;
- and finally the Department of Computer Science of the University of Pretoria for providing the ideal working conditions for undertaking ventures of this kind, and especially Roelf van den Heever for his unfailing encouragement and support.

Judy M Bishop
Organising Chairman, SACRS 1992
Guest Editor, SACJ Special Issue
The journal draws on a wide range of referees. The following were involved in the refereeing of the papers selected for this special issue. Their role in certifying the papers and their contribution to enhancing the quality of papers is sincerely appreciated.

John Barrow 
Ronnie Becker 
Danie Behr 
Sonia Berman 
Liesbeth Botha 
Theo Bothma 
Chris Brink 
Peter Clayton 
Ian Cloete 
Antony Cooper 
Elise Ehlers 
Quintin Gee 
Andy Gravell 
Wendy Hall 
David Harel 
Scott Hazelhurst 
Derrick Kourie 
Willem Labuschagne 
Doug Laing 
Dave Levy 
Graham McLeod 
Hans Messerschmidt 
Deon Oosthuizen 
Ben Oosthuysen 
Neil Pendock 
D Petkov 
Martin Rennhackkamp 
Cees Roon 
Jan Roos 
John Shochot 
Morris Sloman 
Riel Smit 
Pat Terry 
Walter van den Heever 
Lynn van der Vege 
Herman Venter 
Herna Viktor 

UNISA
University of Cape Town
University of Pretoria
University of Cape Town
University of Pretoria
University of Pretoria
University of Cape Town
Rhodes University
Stellenbosch University
CSIR
Rand Afrikaanse University
DALSIG
University of Southampton
University of Southampton
The Weizmann Institute of Science
University of the Witwatersrand
University of Pretoria
UNISA
ISM
University of Natal
University of Cape Town
University of the Orange Free State
University of Pretoria
University of Pretoria
University of the Witwatersrand
University of Natal
Stellenbosch University
University of Pretoria
University of Pretoria
University of the Witwatersrand
Imperial College, London
University of Cape Town
Rhodes University
University of Pretoria
University of Pretoria
University of Fort Hare
Stellenbosch University
Accessing Subroutine Libraries on a Network

P H Greenwood*  
P E Nash†

*INFOTEK, CSIR, P O Box 395, Pretoria, 0001  
†Free Range Computer Systems, P O Box 12475, Onderstepoort, 0110

Abstract

The purchase cost of specialized subroutine libraries can be high, especially if they need to be duplicated on several hosts on a local area network. A remote library access system has been designed, and a prototype built, which allows almost transparent access by a client program to any library residing on any remote host on the network.

Keywords: Remote Invocation, remote procedure call, distributed computing.

Computing Review Categories: C.2.4

Received March 1992, Accepted April 1992

1 Introduction

A key incentive for networking computers together is to enable the sharing of scarce resources, such as a fast computer-server, an expensive software package, or a corporate database. In a typical scientific computing environment, consisting of Unix compute-servers and workstations, PCs, and TCP/IP-based networking services, many facilities are available which promote access to and sharing of resources: rlogin; rsh; Network File System; the X window system. A recent trend is for third-party application software to be licensed on the basis of "floating seats". A single licence manager server on the network allocates licence tokens to users of the software, who may then run the application on any host of their choice on the network.

However, the use of specialized subroutine libraries on a computer network is far more restricted. Usually, a program which makes use of a library will have to run on the host where the library is loaded. Alternatively, several copies of the library must be purchased and placed on all hosts with programs needing access. The latter course is the only one available when a single program needs to call routines in several libraries: all the libraries must reside on a single host.

The apparent necessity of purchasing duplicate copies of subroutine libraries provided the incentive for us to attempt to find a better alternative. The most simplistic approach, namely splitting up a program into several parts, each requiring access to a single library, is usually not an attractive option to the application programmer. We sought a mechanism whereby a programmer could access a subroutine library on a remote host as simply as it would be accessed locally, namely by specifying a library at link time. If possible, no changes to the source code should be needed.

We envisaged developing a Remote Library System (RLS) by which a program on one host could call library routines on a second host. The availability of a high-speed local area network (10 Mbit/s) and robust networking protocols made such a system seem practicable. We had a graphics and a maths subroutine library which we wished, as a start, to access in this manner. However, it was immediately obvious that a general solution, not bound by the idiosyncrasies of any one library, was worth pursuing.

The remainder of this paper is structured as follows. In section 2 we review briefly some prior work. In section 3 we describe the environment in which RLS has to operate and we formulate our design goals. In section 4 we discuss the Sun Remote Procedure Call mechanism and the reasons that it does not meet our requirements. A summary of our experience in designing and implementing a prototype of RLS is given in section 5. In section 6 we summarize our experience of using RLS, including an initial indication of the performance costs.

2 Prior Work

Several working systems have been built which offer some sort of remote execution facilities. The motivation for such work is often provided by the higher performance and greater fault tolerance which distributed computing offers. For example, Stamos et al. [3] describe a remote evaluation construct that involves sending code (source or compiled) to the remote server for execution. This mechanism requires support at compile time. Black et al [1] present a mechanism that allows invocation of operations on application-level objects, where the objects may change their location on the network. No support is needed by the operating system or the programming language. DAWGS [2] is designed to exploit the unused CPU cycles of idle workstations on a network. It provides an interface between the programmer and the kernel that allows interactive and batch-type processes to be executed on remote hosts.

In our case the goal was not to distribute an application over a network for the sake of performance, rather to access a unique resource, namely a subroutine library, from anywhere on the network.
3 Assumptions and Design Goals

There were a number of factors in our computing environment which RLS had to accommodate:

- A heterogeneous computing environment, consisting of computers using different processors and running different versions of Unix;
- An Ethernet network running TCP/IP;
- Scientists who write their own programs, and who are reluctant to accept a computing environment which requires additional effort to use;
- An increasing concern for computer security.

Our design goals followed from the requirement to provide remote library access, and from the above constraints. They were:

- Fully-functional access to remote library routines. There should be no restriction on the type of routines which may be accessed. In particular, subroutines that perform I/O or that have procedures as arguments should be catered for.
- Ease of porting to different flavours of Unix, and to MSDOS. (MSDOS was included to accommodate the possibility of client programs running on PCs.)
- Transparent access by the client program (so as not to burden the application programmer) and to the library (a necessity, as source code for the libraries is usually unavailable to us).
- No reliance on support from compilers or the kernel. We did not have source code for the compilers and operating systems used.
- No additional compromise to the security of the participating hosts.
- Ease of use. For example, it was regarded important that assistance by the system administrator be unnecessary during the development of software making use of RLS.

4 RLS versus Sun RPC

Initially, some thought was given to using the Sun Microsystems Remote Procedure Call (Sun RPC) facility to implement this access. However, the Sun RPC has a few restrictions which make it unsuitable for this application. The major restriction is that function calls may pass a maximum of 4096 bytes of parameters, and that remote procedures may return a maximum of 4096 bytes. Initially, RLS was designed to allow remote access to the DISSPLA and IMSL libraries. Both have routines which often take large arrays of data as input.

A further problem is that Sun RPC calls are stateless. A single server can process calls from a number of clients. However, many of the DISSPLA and IMSL routines set internal variables that affect subsequent behaviour (such as setting the output device for DISSPLA). This is clearly incompatible with Sun RPC. In RLS, each client has its own server process, which serves that client and that client alone. This allows the server state to be maintained between procedure calls in a single session. At the send of a session, the server process is killed.

The Sun XDR (eXternal Data Representation) feature is used by RLS. This provides a network representation of all common data types, and translation routines between the network format and host format, allowing data to be swapped between hosts with different word lengths, byte ordering or floating point formats.

5 Design and Implementation

Overview

A prototype of the system has been built, based on the client-server model. It consists of:

- a general-purpose listener;
- general purpose client and server communication modules; and
- client and server glue modules specific to selected sets of subroutines calls in the DISSPLA and IMSL libraries.

A client program consists of client application code joined by the appropriate client glue module to the client communication module, while a library server consists of the server communication module joined by the appropriate server glue module to the relevant subroutines in the library.

The listener establishes the connection between a client program and the library server. The glue modules provide the interfaces between the client program and the client communications module on the client side, and between the server communications module and the library code on the server side.

The communications modules use a fairly simple protocol layered on top of the error-free data stream provided by the Berkeley sockets transport. This protocol has two components, a simple start-up sequence to sign on a client, and validate the client’s access rights; and a marginally more complex function call-and-return protocol to pass function names, arguments and results.

Initially, a listener waits on the server machine, accepting connections on a socket. The client connects to the socket, and sends a null-terminated request string, giving user name, host computer, library required, password and connection options. The listener validates the user-host-password-library combination, and sends either an accept or a reject message, which is also a null-terminated string. The listener then forks, the parent closes the active socket and goes back to listening. Each client therefore, has a unique copy of the server running.

The child process now does all the work, creating a suitable environment and then executing the relevant library code. Each library has a stub main procedure, linked to the actual function library by suitable glue code. The only routines that may be called by the client are those that are specified by the server glue code and therefore have been statically linked into the server executable. To cope with numeric arguments (different byte orderings, floating point forms, etc), the message format now changes from
The library now loops through a function call and return sequence. The client sends the function name, followed by a list of arguments. The server calls the relevant routine, and returns the return-code followed by the results. To allow values to be returned, all calls into the RLS client code are made by reference. These pointers are dereferenced, and the resulting values are passed to the server. The same happens at the server, which returns the values of the variables, to be placed back into the original client’s memory.

Each argument (either a single value or an array) is preceded by a description string, so that the library routines can invoke the relevant XDR translation procedures, and allocate sufficient storage for processing. These descriptions specify both the type of the argument (byte, int, float, etc) and the size of the array (one for a scalar value). As many numerical routines manipulate large arrays of data, and pass results back in other large arrays, we decided not to pass all arguments back and forth all the time. Instead, each description string specifies whether that particular argument is required by the routine being accessed as an input, output, or input-and-output parameter. This potentially halves network traffic.

Finally, well-behaved client routines will send a shutdown message to the server, so that it can make an orderly shutdown. Few do this, however, so the server has to be able to recognise a dead (or aborted) client, and kill itself gracefully.

Stdin, Stdout and Stderr

Many of the pre-compiled libraries write error and information messages to the stdout and stderr data stream. These need to be merged into the stdout and stderr streams of the calling (client) program, rather than being discarded.

Fortunately, opened Berkeley sockets appear as any other file, which makes redirection easy. As part of the environment creation at start-up, the server opens two connections back to the client. These appear to the server code as ordinary files, so the dup2 Unix system call can connect the function library’s stdout and stderr streams to these sockets.

As sockets are bi-directional, the server’s stdin can be redirected to the same socket as stdout, and any reads of stdin by the library code are reflected back to the client, where they can be serviced either by the client’s program or by the client’s console.

At the client, the output of these sockets can be routed to the client program’s stdout and stderr streams in one of two ways. On multi-tasking operating systems, the client can fork a task which watches these sockets, reading whenever there is data present and writing this data to stdout or stderr. Single-tasking operating systems cannot do this, and on these machines all calls to RLS code will check for data on these sockets, and flush any data to stdout and stderr. Client code can also call the flush function asynchronously.

Other data files

Occasionally, library functions need access to data files on the client computer. These may hold input data or be used to accept results. There are at least three ways in which this can be achieved.

The first, comparatively simple, technique is to copy the file to the server machine using FTP, run the application and then retrieve the file. This is not always feasible, as the file may contain data written by part of the application, and also places a large burden on the user. This would be difficult to automate, as the file is known to RLS only by its Fortran unit number, or Unix file descriptor.

Next, the file could exist on a file-server, and be mounted by both machines. Apart from adding complexity, this reduces the amount of memory available to the client program on DOS-based machines, and runs into problems with buffer flushing if both client and server code access the same file. File naming is also a problem, as described above.

The solution finally chosen was more complex to develop, but is totally transparent to the client and server code. An additional socket connection is created whenever file access is required, and server read and write requests are transmitted down the connection. The client acts as a simple dedicated file-server for this connection, until the remote library function returns, at which point it changes back to normal client-mode operation. For want of a better name, we have called such ancillary channels “tunnels”.

The tunnelling feature was probably the most difficult part of implementation. The end result is not elegant, but it works effectively. The mechanics of synchronising, creating the connection, and keeping data flowing through it were complicated by not having easy access to Fortran internals. Mapping Fortran unit numbers to Unix file descriptors is implementation-dependent, and not always possible. Even were we able to do this, we would still have had to make sure that the Fortran file buffers were flushed, so we opted to have the flushing routine call user-supplied routines to read and write the relevant files. These routines are hidden in the glue code.

Opening a connection

Before a client program can call functions on a remote machine, it must establish the session, as described above. To do this, it must call an open routine. This is usually buried in the stub library on the client machine, so that the client code is unaware of this requirement. The open function must be passed the remote machine’s address (either FQDN or TCP/IP address) and library name.

To allow a single client program to call functions in more than one library, the open call returns a pointer to a data structure (similar to the standard C FILE*), which contains connection information. This pointer is passed to all functions pertaining to that library, such as the call, flush and close functions.

The open function allows for a number of options to be set. At present, we have implemented two. One allows the background flushing of stdout and stderr to be turned on or off (default is off), and the other forces the client to
sleep for 10 seconds when closing a connection, to enable stdout and stderr to flush correctly.

Glue modules
The client glue module provides entry points for all the remote routines required by the client program. These are translated into appropriate calls to the client communications module.

On the server side, the glue module consists of a function dispatcher which is called by the server communications module. The dispatcher determines from its first argument which library routine is being called, and then makes the call with the necessary arguments.

Practical experience
Using Berkeley sockets and Sun XDR made the implementation comparatively painless. It took approximately ten days for one person to implement the communications system, verify that data were moved back and forward correctly, add the tunnel facility and debug the general concept. The glue modules took a further eight days to implement.

It was interesting to see how portable code could be across various hardware and software combinations. The initial code was developed on a 386, running Unix V3.2. This was ported to a R3000-based CDC machine, running a version of EPX without much difficulty. This became the development environment, and the resulting code was finally ported to an IBM RS6000 running AIX 3.1. This final port required a single *ife*de*fe in each of the source files for conditional compilation.

The system has been tested with both CDC and IBM machines as both clients and servers.

6 RLS in Use

RLS has been used by programs requiring mathematical routines (IMSL) and graphics routines (DISPLA). The DISPLA routines are used in both interactive and non-interactive modes. RLS adds an execution time overhead which in a few cases is quite unacceptable, but in many cases is barely perceptible. For example, we tested RLS with a compute-intensive application in which large arrays are passed as parameters by many calls to the remote library. The existing execution time of about an hour was doubled. On the other hand, for applications which made use of simple interactive graphics, the overhead due to RLS was a small fraction of a second for each plot. For programs for which RLS imposes an unacceptably high overhead, there is the option of reducing the amount of RLS network traffic by moving part of the code from the client side to the library server.

It was found to be impractical to build a server for a whole library. Some libraries have several hundred entry points, while a typical program may need access to ten or less. Instead, a library server was built for each client program, or sometimes for a family of related programs.

On the whole, we met the design goal of providing to an application programmer transparent access to a remote library. All that is required of the programmer is to specify a suitable library at link time. This does, however, depend on the necessary glue modules (client and server) having been created, but this need be done only once for any desired set of subroutines. Our goal of transparent access was not met (from the point of view of the application programmer) in the case of routines which have functions as arguments. In this case, it is required of the programmer to implement the function within the library server code, and on the client side, to pass the name of the function as a character string rather than passing its address.

It is relatively straightforward to use RLS. Most of the glue code is generic and easily, if tediously, adapted to a specific set of library routines. The most difficult part is calculating the storage requirements of arguments whose sizes depend on the values of other arguments in the function call.

Several extensions to RLS are being contemplated:
- PC access: in principle, a program running on an MS-DOS PC can access an RLS library by means of a suitable TCP/IP-based interface. Many commercial implementations of TCP/IP for the PC provide a socket interface.
- Glue library specification language and code generator: the glue code for both client and server is eminently suited to being generated by machine. Such a code generator would reduce much of the tedium of building glue modules.
- Distributed processing: RLS provides the building blocks for a poor man's form of distributed processing. A single client could open up connections to many RLS servers, all operating asynchronously with respect to each other. The remote library code executed would be written by the user, so in effect, his application could execute on several processors simultaneously.

7 Conclusions

RLS is a useful tool which, in many cases, provides the application programmer with almost transparent access to libraries on remote hosts on a local area network. No restrictions were found necessary on the type of routines accessed in this way. In many cases the performance overhead is unnoticed, although for programs which make many calls to library routines passing large arguments, the overhead imposed by RLS can dominate execution time. This can be ameliorated by restructuring the client program.

RLS is not a distributed operating system, but it does provide a useful facility for distributing programs across a network of conventional computers. Although all the machines in this implementation of RLS use Unix, there is no reason why any other operating system that has TCP/IP available could not co-operate in this network. This makes RLS a useful vehicle for networks of computers with differing architectures, or for applications where specific hardware and software architectures are required.
References


Acknowledgements: We wish to thank colleagues at the CSIR for stimulating discussions, and for assistance during implementation and testing. Thanks are also due to the reviewers for their constructive comments.
Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal’s readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines:

- Use wide margins and 1 1/2 or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author’s initials and surname;
  - author’s affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author’s surname, and should be cited in the text in square brackets [1, 2, 3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) \LaTeX \documentclass(s), either on a diskette, or via e-mail/ftp – a \LaTeX style file is available from the production editor;
2. In camera-ready format – a detailed page specification is available from the production editor;
3. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

Further instructions on how to reduce page charges can be obtained from the production editor.
4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for \LaTeX or camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 3 and 4 above will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author’s responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

## Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>i</td>
</tr>
<tr>
<td>Referees</td>
<td>ii</td>
</tr>
<tr>
<td>Machine Translation from African Languages to English</td>
<td>1</td>
</tr>
<tr>
<td>DG Kourie, WJ vd Heever and GD Oosthuizen</td>
<td></td>
</tr>
<tr>
<td>Automatically Linking Words and Concepts in an Afrikaans Dictionary</td>
<td>9</td>
</tr>
<tr>
<td>PZ Theron and I Cloete</td>
<td></td>
</tr>
<tr>
<td>A Lattice-Theoretic Model for Relational Database Security</td>
<td>15</td>
</tr>
<tr>
<td>A Melton and S Shenoi</td>
<td></td>
</tr>
<tr>
<td>Network Partitions in Distributed Databases</td>
<td>22</td>
</tr>
<tr>
<td>HL Viktor and MH Rennhackkamp</td>
<td></td>
</tr>
<tr>
<td>A Model for Object-Oriented Databases</td>
<td>27</td>
</tr>
<tr>
<td>MM Brand and PT Wood</td>
<td></td>
</tr>
<tr>
<td>Quantifier Elimination in Second Order Predicate Logic</td>
<td>35</td>
</tr>
<tr>
<td>D Gabbay and HJ Ohlbach</td>
<td></td>
</tr>
<tr>
<td>Animating Neural Network Training</td>
<td>44</td>
</tr>
<tr>
<td>E van der Poel and I Cloete</td>
<td></td>
</tr>
<tr>
<td>HiLOG - a Higher Order Logic Programming Language</td>
<td>53</td>
</tr>
<tr>
<td>RA Paterson-Jones and PT Wood</td>
<td></td>
</tr>
<tr>
<td>ESML - A Validation Language for Concurrent Systems</td>
<td>59</td>
</tr>
<tr>
<td>PJA de Villiers and WC Visser</td>
<td></td>
</tr>
<tr>
<td>Semantic Constructs for a Persistent Programming Language</td>
<td>65</td>
</tr>
<tr>
<td>SB Sparg and S Berman</td>
<td></td>
</tr>
<tr>
<td>The Multiserver Station with Dynamic Concurrency Constraints</td>
<td>75</td>
</tr>
<tr>
<td>CF Kriel and AE Krzesinski</td>
<td></td>
</tr>
<tr>
<td>Mehanizing Execution Sequence Semantics in HOL</td>
<td>81</td>
</tr>
<tr>
<td>G Tredoux</td>
<td></td>
</tr>
<tr>
<td>Statemets - an Alternative Modelling Mechanism for Performance Analysis</td>
<td>87</td>
</tr>
<tr>
<td>L Lewis</td>
<td></td>
</tr>
<tr>
<td>From Batch to Distributed Image Processing: Remote Sensing for Mineral Exploration 1972-1992</td>
<td>95</td>
</tr>
<tr>
<td>N Pendock</td>
<td></td>
</tr>
<tr>
<td>Galileo: Experimenting with Graphical User Interfaces</td>
<td>99</td>
</tr>
<tr>
<td>R Apteker and JM Bishop</td>
<td></td>
</tr>
<tr>
<td>Placing Processes in a Transputer-based Linda Programming Environment</td>
<td>109</td>
</tr>
<tr>
<td>PG Clayton, FK de-Heer-Menlah, EP Wentworth</td>
<td></td>
</tr>
<tr>
<td>Accessing Subroutine Libraries on a Network</td>
<td>117</td>
</tr>
<tr>
<td>PH Greenwood and PH Nash</td>
<td></td>
</tr>
<tr>
<td>A Multi-Tasking Operating System Above MS-DOS</td>
<td>122</td>
</tr>
<tr>
<td>R Foss, GM Rehmet and RC Watkins</td>
<td></td>
</tr>
<tr>
<td>Using Information Systems Methodology to Design an Instructional System</td>
<td>126</td>
</tr>
<tr>
<td>BC O'Donovan</td>
<td></td>
</tr>
<tr>
<td>Managing Methods Creatively</td>
<td>131</td>
</tr>
<tr>
<td>G McLeod</td>
<td></td>
</tr>
<tr>
<td>A General Building Block for Distributed System Management</td>
<td>141</td>
</tr>
<tr>
<td>P Putter and JD Roos</td>
<td></td>
</tr>
</tbody>
</table>