7
Referências Bibliográficas

[1] CHANDY, M.; KESSELMAN, C.. **CC++: A declarative concurrent object oriented programming notation**. Em: Agha, G.; Wegner, P. ; Yonezawa, A., editors, RESEARCH DIRECTIONS IN CONCURRENT OBJECT-ORIENTED PROGRAMMING, p. 218–313. MIT Press, 1993.

[2] BODIN, F.; BECKMAN, P.; GANNON, D.; NARAYANA, S. ; YANG, S.. **Distributed pC++: Basic ideas for an object parallel language**. Em: OONSKI'93 PROCEEDINGS OF THE FIRST ANNUAL OBJECT-ORIENTED NUMERICS CONFERENCE, p. 1–24, abril 1993.

[3] LOVEMAN, D.. **High performance fortran**. IEEE Parallel and Distributed Technology, 1(1):25–42, 1993.

[4] SUNDERMAN, V.. **PVM: A framework for parallel distributed computing**. Concurrency: Practice and Experience, 2(4):315–339, dezembro 1990.

[5] GROPP, W.; LUSK, E. ; SKJELLUM, A.. **Using MPI: Portable Parallel Programming with the Message Passing Interface**. MIT Press, 1994.

[6] CARRIERO, N.; GELERNTER, D.. **Linda in context**. Communications of the ACM, 32(4):444–458, 1989.

[7] BAL, H.; KAASHOEK, M. ; TANENBAUM, A.. **Orca: A language for parallel programming of distributed systems**. IEEE Transactions on Software Engineering, 18(3):190–205, 1990.

[8] ANDREWS, G.; OLSSON, R.. **The SR Programming Language: Concurrency in Practice**. The Benjamin/Cummings Publishing Company, Inc., 1993.

[9] ZAVE, P.. **A compositional approach to multiparadigm programming**. IEEE Software, p. 15–25, setembro 1989.

[10] PAPADOPOULOS, G. A.; ARBAB, F.. **Coordination models and languages**. Em: Zelkowitz, M. V., editor, ADVANCES IN COMPUTERS, volume 46, p. 329–400. Academic Press, agosto 1998.

[11] FOSTER, I.; KESSELMAN, C. ; TUECKE, S.. **The anatomy of the Grid: Enabling scalable virtual organization**. The International Journal of High Performance Computing Applications, 15(3):200–222, Fall 2001.

[12] FOSTER, I.; KESSELMAN, C.. **Globus: A metacomputing infrastructure toolkit**. The International Journal of Supercomputer Applications and High Performance Computing, 11(2):115–128, 1997.

[13] URURAHY, C.. **ALua: Um mecanismo de comunicação em Lua**. Dissertação de mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, setembro 1998.

[14] PFEIFER, A.; URURAHY, C.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Event-driven programming for distributed multimedia applications**. Em: PROCEEDINGS OF SBRC'02 - 20TH BRAZILIAN SYMPOSIUM ON COMPUTER NETWORKS, p. 539–553, Búzios-RJ, Brazil, maio 2002.

[15] URURAHY, C.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **ALua: flexibility for parallel programming**. Computer Languages, 28(2):155–180, dezembro 2002.

[16] IERUSALIMSCHY, R.; FIGUEIREDO, L. H. ; CELES, W.. **Lua—an extensible extension language**. Software: Practice and Experience, 26(6):635–652, 1996.

[17] FIGUEIREDO, L.; IERUSALIMSCHY, R. ; CELES, W.. **Lua: an extensible embedded language**. Dr. Dobb's Journal, 21(12):26–33, dezembro 1996.

[18] PFEIFER, A. L.; URURAHY, C.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **An event-driven system for distributed multimedia applications**. Em: PROCEEDINGS OF DEBS'02 – INTERNATIONAL WORKSHOP ON DISTRIBUTED EVENT-BASED SYSTEMS (HELD IN CONJUNCTION WITH IEEE ICDCS 2002), p. 583–584, Vienna, Austria, julho 2002.

[19] YOUNG, A.; MAGEE, J.. **A flexible approach to evolution of reconfigurable systems**. Em: PROCEEDINGS OF THE IEE/IFIP INTERNATIONAL WORKSHOP ON CONFIGURABLE DISTRIBUTED SYSTEMS, p. 152–163, março 1992.

[20] ACHERMANN, F.; LUMPE, M.; SCHNEIDER, J.-G. ; NIERSTRASZ, O.. **Piccola – a small composition language**. Em: Bowman, H.; Derrick, J., editors, FORMAL METHODS FOR DISTRIBUTED PROCESSING – A SURVEY OF OBJECT-ORIENTED APPROACHES, p. 403–426. Cambridge University Press, 2001.

[21] MAGEE, J.; DULAY, N. ; KRAMER, J.. **Structuring parallel and distributed programs**. Software Engineering Journal, 8(2):73–82, março 1993.

[22] ARBAB, F.; HERMAN, I. ; SPILLING, P.. **An overview of manifold and its implementation**. Concurrency: Practice and Experience, 5(1):23–70, 1993.

[23] GELERNTER, D.; CARRIERO, N.. **Coordination languages and their significance**. Communications of the ACM, 35(2):97–107, fevereiro 1992.

[24] CIANCARINI, P.; ROSSI, D.. **Jada: Coordination and communication for java agents**. Em: SECOND INTERNATIONAL WORKSHOP ON MOBILE OBJECT SYSTEMS: TOWARDS THE PROGRAMMABLE INTERNET (MOS'96), p. 213–228, Linz, Austria, julho 1996. Springer-Verlag.

[25] AHUJA, S.; CARRIERO, N. ; GELERNTER, D.. **Linda and friends**. IEEE Computer, 19(8):26–34, 1986.

[26] LEAL, M.. **LuaTS: Um espaço de tuplas reativo orientado a eventos**. Dissertação de mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, fevereiro 2003.

[27] OUSTERHOUT, J.. **Scripting: Higher level programming for the 21st century**. IEEE Computer, 31(3):23–30, março 1998.

[28] CERQUEIRA, R.. **Um Modelo de Composição Dinâmica entre Sistemas de Componentes de Software**. Tese de Doutorado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, 2000.

[29] FOSTER, I.; KESSELMAN, C.. **The Grid: Blueprint for a New Computing Infrastructure**. Morgan Kaufmann Publishers, Inc., 1999. Capítulo 7.

[30] ALLEN, G.; DRAMLITSCH, T.; FOSTER, I.; GOODALE, T.; KARONIS, N.; RIPEANU, M.; SEIDEL, E. ; TOONEN, B.. **Cactus-G toolkit: Supporting efficient execution in heterogeneous distributed computing environments**. Em: PROCEEDINGS OF 4TH GLOBUS RETREAT, Pittsburgh, PA, 2000.

[31] OUSTERHOUT, J.. **Tcl: An embeddable command language**. Em: PROCEEDINGS OF THE USENIX WINTER 1990 TECHNICAL CONFERENCE, p. 133–146, Berkeley, CA, 1990. USENIX Association.

[32] SCHNEIDER, J.-G.; LUMPE, M. ; NIERSTRASZ, O.. **Agent coordination via scripting languages**. Em: Omicini, A.; Zambonelli, F.; Klusch, M. ; Tolksdorf, R., editors, COORDINATION OF INTERNET AGENTS, p. 153–175. Springer-Verlag, 2001.

[33] DE LIMA, M. J. D.. **ORFEO: Programação Distribuída Orientada a Eventos com Funções e Continuações como Valores de Primeira Classe**. Tese de Doutorado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, julho 2001.

[34] PAPADOPUOLOS, G. A.; ARBAB, F.. **Dynamic reconfiguration in coordination languages**. Em: HPCN EUROPE, p. 197–206, 2000.

[35] RODRIGUEZ, N.; URURAHY, C.; IERUSALIMSCHY, R. ; CERQUEIRA, R.. **The use of interpreted languages for implementing parallel algorithms on distributed systems**. Em: Bougé, L.; Fraigniaud, P.; Mignotte, A. ; Robert, Y., editors, EURO-PAR'96 PARALLEL PROCESSING — SECOND INTERNATIONAL EURO-PAR CONFERENCE, p. 597–600, Volume I, Lyon, France, agosto 1996. Springer-Verlag. (LNCS 1123).

[36] URURAHY, C.; RODRIGUEZ, N.. **ALua: An event-driven communication mechanism for parallel and distributed programming**. Em: PROCEEDINGS OF ISCA 12TH INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS (PDCS-99), p. 108–113, Fort Lauderdale, USA, 1999.

[37] SHAW, M.; GARLAN, D.. **Software Architecture: Perspectives on an Emerging Discipline**. Prentice-Hall, 1996.

[38] BARBOSA, V.. **An Introduction to Distributed Algorithms**. The MIT Press, 1996.

[39] GATES, B.. **VBA and COM**. Byte, 23(3):70–72, março 1998.

[40] BIC, L.; FUKUDA, M. ; DILLENCOURT, M.. **Distributed computing using autonomous objects**. IEEE Computer, 29(8):55–61, agosto 1996.

[41] NEHAB, D.. **LuaSocket: Ipv4 sockets support for the Lua language**. `http://www.tecgraf.puc-rio.br/~diego/luasocket/`.

[42] ANDREWS, G.. **Paradigms for process interaction in distributed programs**. ACM Computing Surveys, 23(1):49–90, 1991.

[43] KAPLAN, C.. **The search for self-documenting code**, 1994. `http://www.cs.washington.edu/homes/csk/paper/`.

[44] HANSEN, P. B.. **Studies in Computational Science: Parallel Programming Paradigms**. Prentice-Hall, 1995.

[45] CORMEN, T. H.; LEISERSON, C. E. ; RIVEST, R. L.. **Introduction to Algorithms**. MIT Press, Cambridge, MA, 1990.

[46] SOUZA, P.. **Asynchronous Organizations for Multi-Algorithm Problems**. Tese de Doutorado, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1993.

[47] LONGO, H.. **Aplicação de *A-Teams* ao problema de recobrimento**. Dissertação de mestrado, Departamento de Ciência da Computação, Universidade Estadual de Campinas, Campinas, SP, Brasil, 1995.

[48] CARDOZO, E.; SICHMAN, S.. **DPSK+P User's Manual – C++ Interface, version 1.0**. FEE/UNICAMP, outubro 1992.

[49] PFEIFER, A. L.. **ALua: Um sistema multi-canal baseado em eventos e mobilidade de código**. Dissertação de mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, julho 2002.

[50] ANDREWS, G.; SCHNEIDER, A.. **Concepts and notations for concurrent programming**, 1983.

[51] LEITE, L.; ALVES, R.; LEMOS, G. ; BATISTA, T.. **DynaVideo – a dynamic video distribution service**. Em: 6TH EUROGRAPHICS WORKSHOP ON MULTIMEDIA (EG MULTIMEDIA 2001), p. 95–106, Manchester, UK, 2001. Springer-Wien.

[52] LANGE, D.; OSHIMA, M.. **Programming and Developing Java Mobile Agents with Aglets**. Addison Wesley, 2000.

[53] IERUSALIMSCHY, R.; CERQUEIRA, R. ; RODRIGUEZ, N.. **Using reflexivity to interface with CORBA**. Em: INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES 1998, Chicago, EUA, 1998. IEEE.

[54] FOSTER, I.; KARONIS, N.. **A grid-enabled mpi: Message passing in heterogeneous distributed computing systems**. Em: PROCEEDINGS OF 1998 SC CONFERENCE, novembro 1998.

[55] DE MOURA, A. L.. **Um ambiente de suporte à adaptação dinâmica de aplicações distribuídas**. Dissertação de mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, 2000.

[56] Object Management Group. **The Common Object Request Broker: Architecture and Specification v2.3.1**, outubro 1999. OMG Document formal/99-10-07.

[57] CERQUEIRA, R.; CASSINO, C. ; IERUSALIMSCHY, R.. **Dynamic Component Gluing Across Different Componentware Systems**. Em: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS (DOA'99), Edinburgh, Scotland, 1999. IEEE Press.

[58] DE MOURA, A.; URURAHY, C.; CERQUEIRA, R. ; RODRIGUEZ, N.. **Dynamic support for distributed auto-adaptive applications**. Em: PROCEEDINGS OF AOPDCS - WORKSHOP ON ASPECT ORIENTED PROGRAMMING FOR DISTRIBUTED COMPUTING SYSTEMS (HELD IN CONJUNCTION WITH IEEE ICDCS 2002), p. 451–456, Vienna, Austria, julho 2002.

[59] **URL do Projeto Globus**.
`http://www.globus.org/`.

[60] CZAJKOWSKI, K.; FOSTER, I.; KARONIS, N.; KESSELMAN, C.; MARTIN, S.; SMITH, W. ; TUECKE, S.. **A resource management architecture for metacomputing systems**. Em: PROCEEDINGS OF IPPS/SPDP '98 WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, p. 62–82, 1998.

[61] CZAJKOWSKI, K.; FITZGERALD, S.; FOSTER, I. ; KESSELMAN, C.. **Grid information services for distributed resource sharing**. Em: PROCEEDINGS OF 10TH IEEE INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE DISTRIBUTED COMPUTING (HPDC-10). IEEE Press, 2001.

[62] **URL do Projeto Grid-Rio**.
`http://easygrid.ic.uff.br/grid/GridRio.html`.

[63] URURAHY, C.; RODRIGUEZ, N.. **Programming and coordinating grid environments and applications**. Em: PROCEEDINGS OF MGC'03 – INTERNATIONAL WORKSHOP ON MIDDLEWARE FOR GRID COMPUTING (HELD IN CONJUNCTION WITH INTERNATIONAL MIDDLEWARE CONFERENCE), p. 261–263, Rio de Janeiro, Brazil, junho 2003.

[64] HOWES, T. A.; SMITH, M. C. ; GOOD, G. S.. **Undestanding and Deploying LDAP Directory Services**. Macmillan Technical Publishing U.S.A., 1999. Macmillan Network Architecture and Development.

[65] CELES, W.. **toLua - accessing C/C++ code from Lua**. `http://www.tecgraf.puc-rio.br/~celes/tolua/`.

[66] BORGES, R.. **Dynamic library loading facilities for the Lua language**, 1998. `http://www.tecgraf.puc-rio.br/~rborges/loadlib/`.

[67] BAGLEY, D.. **The great computer language shootout**, 2001. `http://www.bagley.org/~doug/shootout/`.

[68] FOSTER, I.. **Designing and Building Parallel Programs**. Addison Wesley, 1995.

[69] BATISTA, T. V.. **LuaSpace: Um ambiente para reconfiguração dinâmica de aplicações baseadas em componentes**. Tese de Doutorado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, outubro 2000.

[70] SMITH, B.; ROWE, L. ; YEN, S.. **Tcl distributed programming**. Em: 1ST TCL/TK WORKSHOP, p. 50–51, junho 1993.

[71] GRAY, R.. **Agent Tcl: A transportable agent system**. Em: PROCEEDINGS OF THE CIKM WORKSHOP ON INTELLIGENT INFORMATION AGENTS, FOURTH INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT (CIKM 95), dezembro 1995.

[72] SCHUMACHER, M.; CHANTEMARGUE, F. ; HIRSBRUNNER, B.. **The STL++ coordination language: A base for implementing distributed multi-agent applications**. Em: PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON COORDINATION MODELS AND LANGUAGES, p. 399–414. Springer-Verlag, 1999.

[73] PAXSON, V.; SALTMARSH, C.. **Glish: A user-level software bus for loosely-coupled distributed systems**. Em: 1993 WINTER USENIX TECHNICAL CONFERENCE, p. 141–156, 1993.

[74] VON LASZEWSKI, G.; FOSTER, I. ; GAWOR, J.. **CoG kits: A bridge between commodity distributed computing and high-performance grids**. Em: JAVA GRANDE, p. 97–106, 2000.

[75] RANA, O. F.; WALKER, D. W.. **'The agent grid': Agentbased resource integration in PSEs**. Em: PROCEEDINGS OF THE 16TH IMACS WORLD

CONGRESS ON SCIENTIFIC COMPUTATION, APPLIED MATHEMATICS AND SIMULATION, Lausanne, Switzerland, agosto 2000.

A
API do ALua

Apesar do ALua oferecer uma única primitiva de comunicação (`send`), o sistema oferece outras funções para dar suporte à criação e finalização de processos, facilidades de comunicação e outras tarefas.

Todas as funções do ALua descritas a seguir (e algumas variáveis) são acessadas através de uma tabela chamada `alua`, ou seja, é necessário usar o identificador `alua` antes do nome da função. Por exemplo, a função `spawn` é acessada através do comando: `alua.spawn(...)`.

Esta documentação, assim como exemplos de código e publicações, estão disponíveis em:

`http://www.inf.puc-rio.br/~ururahy/alua/`

A.1
API Básica

---

spawn

- **Interface**: alua.spawn(nproc, spawn_completed [, execfile, desthosts])
- **Description**: Start processes to be used in the application.
- **Input parameters**:

  1. nproc: number of processes to be spawned. This parameter can be a number or a table with the name of the processes.
  2. spawn_completed: callback function to be called after all processes are configurated and ready to send and receive messages. When this function

is called, the table holding the processes created in the spawn function is passed as a parameter.

3. execfile: file to be executed when the new process is created by the daemon. If this parameter is not available, the daemon will exec the file **agent** (located in the directory $ALUADIR).
4. desthosts: hosts where the processes will be created. If this parameter is not available, the processes will be created in hosts listed in file **aluacfg.lua** (located in the directory $ALUADIR).

– **Output parameters**: none

– **Comments**:

1. Each spawned process will receive the following variables:
   – alua.mytid: its index in the processes list passed as a parameter to function spawn_completed
   – alua.myname: process name;
   – alua.myhost: process host name;
   – alua.myparent: name of the parent process.
   – alua.myparenthost: name of the parent process host.
   – alua.mysession: number of my session.
2. When a process executes the spawn function, the ALua environment keep the name of the spawned processes (children processes) in an internal table. This table is used in functions like broadcast and exit_all, when all these children processes will be affected.
3. The variables alua.myparent and alua.myparenthost are not defined for the master process (the first process created when the command run is executed).

---

send_agentid

– **Interface**: alua.send_agentid(to, tb)

– **Description**: In ALua, when the processes are created, they don't know each other and they can not send code messages to other processes (except to **master process** - the first process created by ALua when the command run is executed). However, sometimes it is necessary that the processes know each other. To solve this problem, the user can use the function alua.send_agentid that sends the identification of the processes in table **tb** to all processes listed in **to**.

- **Input parameters**:
  1. to: name of the process(es) to send the identifications (can be a string or a lua table)
  2. tb: table with names of the processes to be identified
- **Output parameters**: none
- **Comments**: This function is typically called in the spawn_completed function when the user desires all processes know each other.

---

send

- **Interface**: alua.send(to, msg)
- **Description**: Send the message **msg** to processes **to**.
- **Input parameters**:
  1. to: destination of the message. This parameter can be a process name or a communication channel.
  2. msg: can be a lua chunck of code or another type of message if the **to** parameter is a communication channel.
- **Output parameters**: none
- **Comments**: none

---

mcast

- **Interface**: alua.mcast(proctab, msg)
- **Description**: Send the message **msg** to all processes in table **proctab**.
- **Input parameters**:
  1. proctab: lua table with the name of the processes to send the message **msg**
  2. msg: lua chunck of code
- **Output parameters**: none
- **Comments**: none

---

broadcast

- **Interface**: alua.broadcast(msg)
- **Description**: Send the message **msg** to all children processes (processes spawned by the process executing the command).
- **Input parameters**:
  1. msg: lua chunck of code
- **Output parameters**: none
- **Comments**:

---

ping

- **Interface**: alua.ping(p)
- **Description**: Send a ping message to the desired process. If the target process is alive it will send back the following message: **process <process name> is alive**.
- **Input parameters**:
  1. p: target process.
- **Output parameters**: none
- **Comments**: none

---

exit

- **Interface**: alua.exit(p)
- **Description**: Stops the specified processes execution.
- **Input parameters**:
  1. p: this parameter can be a table, a string or nothing. If it is a table, it must contain the names of the processes to be stopped. A string must contain the name of process to be stopped. If it is not available, the current process will be stopped.

- **Output parameters**: none.
- **Comments**: none.

---

exit_all

- **Interface**: alua.exit_all()
- **Description**: Stops the execution of all processes spawned through function alua.spawn.
- **Input parameters**: none
- **Output parameters**: none
- **Comments**: none

---

insert

- **Interface**: alua.insert(command)
- **Description**: Inserts a command in the transmission buffer, which can can be transmited to one or more processes through functions alua.mcast_buffer and alua.send_buffer.
- **Input parameters**:
  1. command: command to be inserted in buffer
- **Output parameters**: none
- **Comments**:
  1. alua.insert and alua.insertvar are functions that deal with transmission buffers, that is, when one of this function is called, their messages are not transmitted immediately, they are kept in an ALua message buffer. The commands kept in the message buffer are sent just when the user calls alua.mcast_buffer or alua.send_buffer.
  2. The transmission buffer keeps up to 4096 bytes.

---

insertvar

- **Interface**: alua.insertvar(varname, varvalue)
- **Description**: This function creates an attribution command of type "varname = varvalue" and inserts it in the ALua transmission buffer, which can be transmited to one or more processes through functions alua.mcast_buffer and alua.send_buffer.
- **Input parameters**:
  1. varname: name of the variable in the target process.
  2. varvalue: variable's value.
- **Output parameters**: none
- **Comments**: see also alua.insert

---

mcast_buffer

- **Interface**: alua.mcast_buffer(proctab, msg)
- **Description**: Sends the transmission buffer contents to the target processes.
- **Input parameters**:
  1. proctab: table with the target processes.
  2. msg: message printed when ALua starts the transmission of the buffer contents.
- **Output parameters**: none
- **Comments**: none

---

send_buffer

- **Interface**: alua.send_buffer(to, msg)
- **Description**: Sends the transmission buffer contents to the target process.
- **Input parameters**:
  1. to: name of the target process.
  2. msg: message printed when ALua starts the transmission of the buffer contents.
- **Output parameters**: none
- **Comments**: none

A.2
API Estendida

---

new_srvchannel

- **Interface**: alua.new_srvchannel(port, readFunc, writeFunc, connFunc, closeFunc)
- **Description**: Creates a new TCP server socket on the local host (see also bind).
- **Input parameters**:
  1. port: Port to which the new server must be bound. You also must set the address. The function sethostname is used to do this. If the address is not configured, ALua will bind the new TCP server socket to "localhost".
  2. readFunc: Callback invoked whenever the channel is ready for reading.
  3. writeFunc: Callback invoked whenever the channel is ready for writing.
  4. connFunc: Callback invoked whenever a new channel is established.
  5. closeFunc: Callback invoked whenever the channel is about to be closed.
- **Output parameters**: In case of success, the function returns a server socket. In case of error, the function returns nil followed by a string describing the error.
- **Comments**:
  1. The readFunc, writeFunc and closeFunc are never triggered in the server channel. They are automatically defined as callbacks for the new communication channels established with the server channel.
  2. By default the ALua environment calls a receive function when a channel is ready for reading. The data received, along with the socket, is then passed as parameters to the callback associated with read events. The user can change this behavior setting the attribute **mustReceive** of the server channel to **nil** after creating the server channel. If the user chooses call the receive function, the readFunc will be triggered when the channel is ready for reading and the socket will be passed as a parameter. The user can then call the receive function to get the data. It is important to use the receive function only in these cases, otherwise the system can be

blocked waiting for data in the communication channel and not process upcoming events.

---

new_channel

- **Interface**: alua.new_channel(host, port, readFunc, writeFunc, closeFunc)
- **Description**: Creates a client connection to a TCP server (see also bind).
- **Input parameters**:
  1. host: Host to which the client will try to connect.
  2. port: Port to which the client will try to connect.
  3. readFunc: Callback invoked whenever the new channel is ready for reading.
  4. writeFunc: Callback invoked whenever the new channel is ready for writing.
  5. closeFunc: Callback invoked whenever the new channel is about to be closed.
- **Output parameters**:
- **Comments**:

---

set_handler

- **Interface**: alua.set_handler(soc, mode, func)

  Description: Allows the programmer to change the callbacks associated with a given communication channel at any point of execution.
- **Input parameters**:
  1. soc: socket table representing the communication channel.
  2. mode: indicate which event will be changed. It can be “Read”, “Write”, “Connect” or “Close”.
  3. func: callback to be associated with the event. If this parameter is nil, the event “mode” will no longer be monitored.
- **Output parameters**: In case of error, a string describing the error is returned.

- **Comments**:

---

close_channel

- **Interface**: alua.close_channel(soc)
- **Description**: Close the connection of a communication channel.
- **Input parameters**:
  1. soc: socket table.
- **Output parameters**: none
- **Comments**:
  1. This function will trigger the closeFunc function associated with the communication channel passed as a parameter.

---

add_function (Verificar parâmetros...)

- **Interface**: alua.add_function(peerip, peerport, funcName, func)
- **Description**: Associate a function with an existing channel in a remote process.
- **Input parameters**:
  1. peerip: IP address of the peer connection.
  2. peerport: Port of the peer connection.
  3. funcName: name of the function to be associated with the channel.
  4. func: function to be associated with the channel.
- **Output parameters**: In case of error, a string describing the error is returned.
- **Comments**:

---

sethostname

- **Interface**: alua.sethostname(hostname)

- **Description**: Set the host name. This name is used in the function alua.new_srvchannel to bind the new TCP server socket. If this name is not defined, the alua.new_srvchannel will bind to "localhost".
- **Input parameters**:
  1. hostname: name of the host.
- **Output parameters**: none
- **Comments**: none

---

event_manager

- **Interface**: alua.event_manager()
- **Description**: Execute the event loop checking for incoming events.
- **Input parameters**: none
- **Output parameters**: none
- **Comments**:
  1. ALua automatically calls this function. However, you can also create an isolated process and associate it with an ongoing session (see also associate_proc). In this case, you will have to call this function to check for ALua events.

---

associate_proc

- **Interface**: alua.associate_proc(session, procname, prochost, parentname, parenthost)
- **Description**: Associates a process with an ongoing session.
- **Input parameters**:
  1. session: session with which the process will be associated.
  2. procname: name of the process in the session.
  3. prochost: host of the process (where the process is executing).
  4. parentname: name of the parent process.

5. parenthost: host of the parent process.

– **Output parameters**: none

– **Comments**:

1. The ALua multichannel communication infrastructure enables that multiple computations run at the same time. To avoid cross messages between processes running in different computations a session number identifies each computation. This number is unique to each computation. When a new process is created, it is necessary to associate this process with an ongoing session if it desires to exchange message with other processes in the same session. This function associates a new process with an ongoing session. The session is created when the run command is executed and the master process is created.

---

new_udpsrv

– **Interface**: alua.newudpsrv(port, readFunc, writeFunc)

– **Description**: Creates a new UDP server socket on the local host (see also udpsocket).

– **Input parameters**:

1. port: Port to which the new server must be bound. You also must set the address. The function sethostname is used to do this. If the address is not configured, ALua will bind the new TCP server socket to "localhost".
2. readFunc: Callback invoked whenever the channel is ready for reading.
3. writeFunc: Callback invoked whenever the channel is ready for writing.

– **Output parameters**: In case of success, the function returns a server socket. In case of error, the function returns nil followed by a string describing the error.

– **Comments**: none

---

new_udpclt

– **Interface**: alua.new_udpclt(readFunc, writeFunc)

- **Description**: Creates a client UDP socket (see also udpsocket).
- **Input parameters**:
  1. readFunc: Callback invoked whenever the new socket is ready for reading.
  2. writeFunc: Callback invoked whenever the new socket is ready for writing.
- **Output parameters**: In case of success, the function returns a server socket. In case of error, the function returns nil followed by a string describing the error.
- **Comments**: none

B
API do ALuaMonitor

B.1
Funções do ALuaMonitor

---

defineProperty

- **Interface**: ALuaMonitor:defineProperty(propname, getf, freq)
- **Descrição**: Define uma propriedade a ser monitorada.
- **Parâmetros de entrada**:
  1. propname: nome que identifica a propriedade.
  2. getf: função que atualiza o valor corrente da propriedade.
  3. freq: freqüência com a qual o valor da propriedade será atualizado.
- **Parâmetros de saída**: —
- **Comentários**: O mecanismo do ALuaMonitor se encarrega de chamar a função `getf` para atualizar o valor da propriedade a cada `freq` segundos.

---

destroyProperty

- **Interface**: ALuaMonitor:destroyProperty(prop)

  Descrição: Retira a propriedade `prop` da lista de propriedades monitoradas pelo ALuaMonitor.
- **Parâmetros de entrada**:

1. prop: nome que identifica a propriedade que será removida da lista de propriedades.

– **Parâmetros de saída**: —

– **Comentários**: —

---

attachEventObserver

– **Interface**: ALuaMonitor:attachEventObserver(obj,evID,notifyCond)

– **Descrição**: Cria um observador para o evento identificado por `evID`.

– **Parâmetros de entrada**:

1. obj: um objeto Lua, representado por uma tabela, que deve conter pelo menos a função `notifyEvent`.
2. evID: nome do evento.
3. notifyCond: *string* com a expressão booleana que determina a ocorrência do evento em questão. Os valores das propriedades usados na expressão booleana devem ser descritos como `$<nome da propriedade>` e serão traduzidos para `<propriedade>:getValue()`. Um observador não está amarrado a uma única propriedade e pode fazer uso de um conjunto delas para determinar um evento que se deseja observar. Para ser notificado quando a propriedade `X` for maior que 5 e a propriedade `Y` for igual a 2, basta criar um observador que espera pelo evento `"$X > 5 and $Y == 2"`.

– **Parâmetros de saída**: —

– **Comentários**: A função `notifyEvent` recebe o nome do evento como único parâmetro.

Um exemplo de objeto observador é dado a seguir:

```
observer = {
   notifyEvent = function(self, event_name)
      alua.send("A", "greaterThanThree()")
   end
}
```

---

detachEventObserver

- **Interface**: ALuaMonitor:detachEventObserver(evID)
- **Descrição**: Remove o observador definido por `evID` da lista de observadores definida no ALuaMonitor.
- **Parâmetros de entrada**:
  1. evID: *string* que identifica o evento observado.
- **Parâmetros de saída**: —
- **Comentários**: —

---

existsEventObserver

- **Interface**: ALuaMonitor:existsEventObserver(evID)
- **Descrição**: Verifica a existência de um observador definido por `evID`.
- **Parâmetros de entrada**:
  1. evID: *string* que identifica o evento observado.
- **Parâmetros de saída**: `1`, caso o observador exista, `nil` caso contrário.
- **Comentários**:

B.2

Funções de um objeto propriedade (Property)

---

defineAspect

- **Interface**: Property:defineAspect(name,updatef)
- **Descrição**: Define um aspecto de uma propriedade. A definição de um aspecto pode ser baseada em uma estatística, em perfis da evolução de alguma condição, etc.
- **Parâmetros de entrada**:
  1. name: nome que identifica o aspecto definido.
  2. updatef: função que atualiza o valor do aspecto definido.

- **Parâmetros de saída**: —
- **Comentários**: O mecanismo do ALuaMonitor se encarrega de chamar a função `updatef` para atualizar o valor do aspecto quando necessário.

---

getAspectValue

- **Interface**: Property:getAspectValue(name)
- **Descrição**: Retorna o valor do aspecto identificado por `name`.
- **Parâmetros de entrada**:
  1. name: *string* que identifica um aspecto.
- **Parâmetros de saída**: Pode ser uma tabela, um número, uma string, etc. Será o valor calculado (e retornado) pela função `updatef`. Caso o aspecto de identificado por `name` não exista, a função retorna `nil`.
- **Comentários**: —

---

definedAspects

- **Interface**: Property:definedAspects()
- **Descrição**: Retorna uma tabela com o nome de todos os aspectos definidos.
- **Parâmetros de entrada**: —
- **Parâmetros de saída**: Tabela com o nome de todos os aspectos definidos.
- **Comentários**: —

---

getValue

- **Interface**: Property:getValue()
- **Descrição**: Retorna o valor corrente de uma propriedade.
- **Parâmetros de entrada**: —
- **Parâmetros de saída**: O valor corrente de uma propriedade. Pode ser uma tabela, um número, uma string, etc. Será o valor calculado (e retornado) pela função `getf`, definida na criação da propriedade (ver `defineProperty`).

- **Comentários**: —

---

destroy

- **Interface**: Property:destroy()
- **Descrição**: Retira uma propriedade da lista de propriedades monitoradas pelo ALuaMonitor.
- **Parâmetros de entrada**: —
- **Parâmetros de saída**: —
- **Comentários**: —

C
Listagem de Códigos

C.1
Capítulo 4

C.1.1
Servidor de Arquivos (C)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <unistd.h>
#include "errexit.h"
#include "mysocket.h"

#define BLOCKSIZE 8192

/*------------------------------------------------------------------------
 * main
 *------------------------------------------------------------------------
 */
int main(int argc, char* argv[])
{
   FILE  *fp;
   char   buffer[BLOCKSIZE];
   size_t nbytes;

   int asock;
   int ssock;
   struct sockaddr target;
   int targetsize=sizeof(struct sockaddr);
```

```
   int port=atoi(argv[1]);

   asock = passiveTCP(&port, 10, 0);
   while (1) {
      ssock = accept(asock, &target, &targetsize);
      if (ssock < 0){
         printf("ERRO = %d\n", errno);
         errexit("ERROR: (accept failed) %s\n", sys_errlist[errno]);
      }

      fp = fopen(argv[2], "r");
      nbytes = fread((void *)buffer,  1, BLOCKSIZE, fp);
      while (nbytes > 0)
      {
         int nsent=0;
         do {
            int ret = write(ssock, buffer+nsent, nbytes-nsent);
            if (ret < 0)
               errexit("ERROR: (write error) %s\n", sys_errlist[errno]);
            nsent += ret;
         }
         while (nsent < nbytes);
         nbytes = fread((void *)buffer,  1, BLOCKSIZE, fp);
      }
      fclose(fp);
      close(ssock);
   }
   close(asock);
   return 0;
}
```

C.1.2

Download Simples C

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "mysocket.h"

#define BLOCKSIZE 8192
#define FILENAME  "/tmp/outputC"

/*----------------------------------------------------------------------
 * main
 *----------------------------------------------------------------------
 */
int main(int argc, char* argv[])
{
   int  asock;
   char host_name[]="localhost";
   int  port=atoi(argv[1]);

   FILE  *fp = fopen(FILENAME, "w");
   char   buffer[BLOCKSIZE];
   size_t nbytes;

   asock = connectTCP(host_name, port);

   do {
      /* nbytes=read(asock, buffer, BLOCKSIZE); */
      nbytes=recv(asock, buffer, BLOCKSIZE, 0);
      fwrite((void *)buffer,  1, nbytes, fp);
   } while (nbytes > 0);

   fclose(fp);
   close(asock);
   return -1;
}
```

C.1.3

Download Simples Lua

```
PORT = 6067
FNAME  = "/tmp/outputluasocket"
BLOCKSIZE = 2^13  -- 8K

function main()
   local sock
   local host_name="server.par.inf.puc-rio.br"
   local n=0
   writeto(FNAME)
   sock = connect(host_name, PORT)

   local buffer, err
   repeat
      buffer, err = receive(sock, BLOCKSIZE)
      if buffer then
         write(buffer)
      end
      n = n + BLOCKSIZE
   until err
   if err ~= "closed" then
      print("ERROR", err)
   end
   writeto()
   close(sock)
   print("Fim")
end

main()
```

C.1.4

Download Simples ALua

```
FILENAME  = "/tmp/outputALua"
BLOCKSIZE = 2^13    -- 8K
HOSTNAME  = "server.par.inf.puc-rio.br"
PORT      = 6067

function final()
   print("Fim")
   alua.close_channel(mychannel)
   writeto()  -- close output file
   exit()
end

do
   local readFunc = function(ch)
      local buffer, err = luasocket.receive(ch, BLOCKSIZE)
      if buffer then
         write(buffer)
      end
      if err then
         if err == "closed" then alua.send(alua.myname,"final()")
         else error(err)
         end
      end
   end

   writeto(FILENAME)
   mychannel = alua.new_channel(HOSTNAME, PORT, readFunc, nil)
   mychannel.mustReceive = nil
end
```

C.1.5

Download Concorrente C (Multithread)

```
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <unistd.h>
#include "mysocket.h"

#define BLOCKSIZE 8192
#define N 3

char* FNAME[] = {"/tmp/output00",
                 "/tmp/output01",
                 "/tmp/output02"};
char* HNAME[] = {"n00.par.inf.puc-rio.br",
                 "n01.par.inf.puc-rio.br",
                 "n02.par.inf.puc-rio.br"}
int port;

void* f(void* args) {
   int n = *(int*)args;
   int  asock;

   FILE  *fp = fopen(FNAME[n], "w");
   char   buffer[BLOCKSIZE];
   size_t nbytes;

   asock = connectTCP(HNAME[n], port);

   do {
      /* nbytes=read(asock, buffer, BLOCKSIZE); */
      nbytes=recv(asock, buffer, BLOCKSIZE, 0);
      fwrite((void *)buffer,  1, nbytes, fp);
   } while (nbytes > 0);

   fclose(fp);
   close(asock);
   return NULL;
}

/*--------------------------------------------------------------------
 * main
 *--------------------------------------------------------------------
```

```
 */
int main(int argc, char** argv)
{
   int i;
   pthread_t thread[N];
   int nthreads = N;
   int n[N];

   port=atoi(argv[1]);
   if (argc >= 3) nthreads = atoi(argv[2]);

   for (i=0; i < N; i++)
   {
      n[i]=i;
      if (pthread_create(&thread[i], NULL, f, (void*)&n[i]))
         printf("ERROR\n");
   }
   for (i=0; i < N; i++)
      pthread_join(thread[i], NULL);

   return 0;
}
```

C.1.6

Download Concorrente ALua

```
N = 3
FNAME  = {"/tmp/outputMCn00",
          "/tmp/outputMCn01",
          "/tmp/outputMCn02",
         }
HOSTNAME  = {"n00.par.inf.puc-rio.br",
             "n01.par.inf.puc-rio.br",
             "n02.par.inf.puc-rio.br",
            }

BLOCKSIZE = 2^13    -- 8K
PORT      = 6068

function final()
   nchannels = nchannels or 0
   nchannels = nchannels + 1
   if nchannels == N then
      exit()
   end
end

do
   local readFunc = function(ch)
      local fd=ch.fdescr
      local buffer, err = alua.receive(ch, BLOCKSIZE)
      if buffer then
         write(fd, buffer)
      end
      if err then
         if err == "closed" then
            writeto(fd)  -- close output file
            alua.close_channel(ch)
            alua.send(alua.myname,"final()")
         else error(err)
         end
      end
   end

   mychannel={}
   for i=1,N do
      mychannel[i] = alua.new_channel(HOSTNAME[i], PORT, readFunc, nil, nil)
      if mychannel[i] then
         mychannel[i].mustReceive = nil
```

```
            mychannel[i].fdescr = writeto(FNAME[i])
         else
            print("Could not create channel.")
         end
      end
end
```

C.2

Capítulo 5

C.2.1

Aplicação MPI Mestre/Escravo

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

#define MAX     2000
#define N       3
#define AMNT    50

#define MASTER  0

#define TASK      0
#define TERMINATE 1

int ntasks = 0;
int amount = AMNT;
int pending_msg = 0;

void nextTask(int slave)
{
   int bounds[2];
   bounds[0] = ntasks + 1;       // first
   bounds[1]  = ntasks + amount; // last
   if (bounds[1] > MAX) bounds[1] = MAX;
   if (bounds[0] <= MAX) {
      MPI_Send(bounds,          /* buff  */
               2,               /* count */
               MPI_INT,         /* type  */
               slave,           /* dest  */
               TASK,            /* tag   */
               MPI_COMM_WORLD); /* comm  */
      pending_msg++;
      printf("Master: Sending tasks %d/%d to slave %d\n",
             bounds[0], bounds[1], slave);
   }
   ntasks = bounds[1];
}
```

```
int main(int argc, char **argv)
{
    int numprocs, my_id;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

    if (numprocs > 1)
    {
        if (my_id == MASTER)
        {
           int i;
           int received_num;
           int slave;
           for (i=1; i < numprocs; i++)
              nextTask(i);
           while ((ntasks < MAX) || (pending_msg))
           {
              MPI_Recv(&received_num,    /* buff   */
                       1,                /* count  */
                       MPI_INT,          /* type   */
                       MPI_ANY_SOURCE,   /* source */
                       MPI_ANY_TAG,      /* tag    */
                       MPI_COMM_WORLD,   /* comm   */
                       &status);         /* status */
              pending_msg--;
              slave=received_num;
              nextTask(slave);
           }

           /* Tell slaves to terminate! */
           for (i = 1; i < numprocs; i++)
           {
              MPI_Send(&i,               /* buff  */
                       2,                /* count */
                       MPI_INT,          /* type  */
                       i,                /* dest  */
                       TERMINATE,        /* tag   */
                       MPI_COMM_WORLD); /* comm  */
           } /* endfor */
        }
        else
        {
           /* I am a Slave */
           int terminate=0;
```

```
            int bounds[2];

            while (!terminate)
            {
                MPI_Recv(bounds,        /* buff   */
                        2,               /* count  */
                        MPI_INT,         /* type   */
                        MASTER,          /* source */
                        MPI_ANY_TAG,     /* tag    */
                        MPI_COMM_WORLD,  /* comm   */
                        &status);        /* status */

                terminate = (status.MPI_TAG == TERMINATE);
                if (!terminate)
                {
                   int i;
                   for (i=bounds[0]; i <= bounds[1]; i++)
                      printf("Doing task %d\n", i);

                   MPI_Send(&my_id,       /* buff  */
                           1,                  /* count */
                           MPI_INT,            /* type  */
                           status.MPI_SOURCE,  /* dest  */
                           TASK,               /* tag   */
                           MPI_COMM_WORLD);    /* comm  */
                }
            } /* endwhile */
        } /* endif */
    }
    else
        printf("numprocs = %d, should be run with numprocs > 1\n", numprocs);

    MPI_Finalize();
    printf("Terminated %d...\n", my_id);

    exit(0);
} /* end main() */
```