On the Adequate Support of Communication Interfaces in Distributed Systems

Prof. Dr. J. Nehmer
Universität Kaiserslautern, Fachbereich Informatik
Erwin-Schrödinger-Str., 6750 Kaiserslautern

Abstract

Existing experimental distributed systems usually support a single mechanism for message-based communication. It is argued that different needs at the operating system level and at the application level make it highly desirable to support several, possibly incompatible communication interfaces in the same distributed system.The resulting problems are investigated and appropriate architectural solutions are proposed. As an example we demonstrate how the two different distributed languages LADY and CSSA with their distinct communication models are implemented in the INCAS project and discuss some problems encountered during the system design.

Key Phrases
Distributed system, message communication, distributed operating system kernel

1. Introduction

Existing research projects on distributed systems are usually based on a single communication model for structuring distributed programs. The communication model is supported at run time by a kernel which offers a suitable set of communication primitives implementing the model. These primitives are made accessible to users either by the provision of library interface packages as in Demos-MP [MIL87], Locus [WAL83], Amoeba [MUL84], and the V-kernel [CHE84] or by a distributed programming language as for example in Eden [BLA85, ALM85], Cedar [SWI85], Argus [LIS83], SR [AND82], NIL [STR85], Linda [CAR85], and Lynx [SCO88].

This approach is based on the assumption that the distributed operating system and the various distributed applications running on top of the kernel can make efficient use of the same communication mechanism. According to our opinion based on experiences within the INCAS project this assumption is generally not true.

Distributed operating systems and distributed applications usually have specific communication requirements which might not be compatible with each other. Careful design considerations concerning the support of the various communication interfaces in distributed systems are necessary in order to avoid system misconceptions leading to severe performance degradations and / or loss of desired functionality.

Relative little attention has been spent by researchers to address this issue. Scott [SCO86] discusses in his paper the related problem of appropriate support of high-level distributed programming languages by distributed operating system kernels. By three implementations of the language Lynx [SCO87] on different distributed operating system kernels he could show that simple communication primitives provided by the kernel are best. However, the interface problem between distributed applications and the distributed operating system is not addressed in the paper. In Accent [RAS81, FIT86] the support of multiple distributed programming languages was an explicit design goal but restricted to RPC-based communication models.

This paper is organized as follows: In section two we classify the communication interfaces in distributed systems. In section three we provide a framework for architectural solutions based on varying communication requirements for distributed operating systems and distributed applications. In section four it is discussed how two rather different communication models at the distributed OS level and the application level are realized in the INCAS project based on the two different languages LADY and CSSA. Section five gives an overview of the problems encountered during the system design of INCAS. The final section six discusses the possible lessons to be learned and summarizes our conclusions.

2. The communication interfaces in distributed systems

Let us take a deeper insight into the different types of communication interfaces we are generally faced with in distributed systems. It is assumed that the distributed operating system and the distributed applications consist of multiple communicating modules called operating system modules (OSM's) and application program modules (APM's).

From Fig. 1 we can identify three interface types:
A : interface between different APM's
B : interface between APM's and OSM's
C : interface between different OSM's

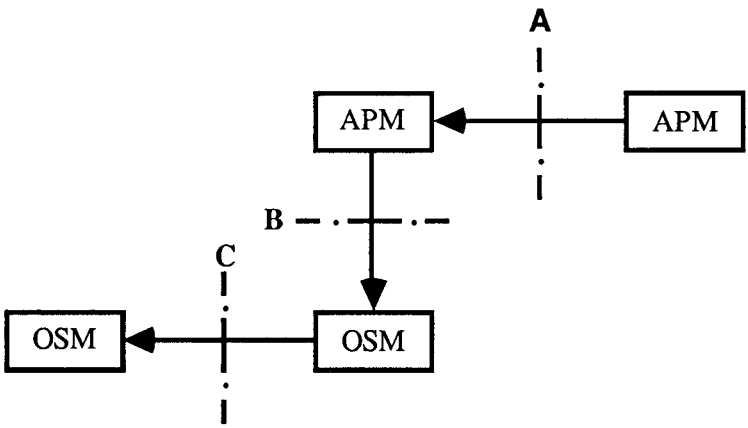

Fig. 1 Interface types between application program modules (APM's) and operating system modules (OSM's)

All three interface types will result in specific requirements on the underlying communication model supporting controlled communication between modules of a given type. The potential of modules operating as active and independent units on different processing nodes offers a broad variety of possible assessments for communication interfaces. At present no consensus on a consistent and comprehensive classification has been reached in the research community. Useful attempts as the basis for further discussion have been contributed by Shatz [SHA84], Jul [JUL85], and Liskov [LIS85]. For the following discussions we will use a classification scheme for communication models which takes into account the three design parameters

- synchrony
- communication pattern
- reliability

The parameter 'synchrony' may obtain the values 'synchronous' and 'asynchronous'. The parameter 'communication pattern' may obtain the values 'notification' and 'service'. Notification-based communication patterns support a one-way communication as used in producer-consumer type relations between communicating modules. Service-based communication supports the request /reply paradigm as needed for client/server systems. The reliability parameter may take the values 'don't care', 'at-least-once', 'at-most-once', 'all-or-nothing', and 'exactly once'.

While it is difficult to recommend certain structures for the application-dependent interface type A it seems more promising to define precise requirements for the interface types B and C because they are devoted to the well known scope of operating systems. However, this assumption is in contradiction to the reached consensus on adequate structuring models for distributed operating systems. Different structuring philosophies (process/message paradigm as opposed to object/atomic action paradigm) and the taste of designers for what is felt to be important have led to rather incompatible proposals for communication models at the distributed operating system level. See for example the different views taken in the languages SR [AND82], NIL [STR85], EPL [BLA85] and LADY[NEH87] which emphasize operating /communication systems as the application scope.

In order to simplify the discussion we make the reasonable assumption that the communication mechanism provided for the interaction between APM's and OSM's (interface type B) is a subset of the mechanism provided for communication within the operating system itself (interface type C), i.e. B⊂C. Traditionally, the communication at the interface type B is restricted to a synchronous, service-oriented call. RPC-like mechanisms [BIR85] are a sufficient realization basis. The requirement stated above means that any communication model at the interface type C includes the support of a synchronous, service directed call. Within the distributed operating and the distributed applications system it might be necessary to provide additional communication primitives for the easy realization of pipelined and multicast/broadcast communication structures.

3. A discussion of systematic architectural alternatives

We now discuss various alternatives for the communication interface types A, B, and C with respect to the architectural support needed. As the general architectural model we base our considerations on the distributed kernel approach. We assume that a distributed kernel provides the functional support for the creation/termination of communicating modules (processes, process groups) and the system-wide communication between them. Operating system services and application programs are both organized as modules running above the kernel.

Alternative 1 : A = B = C

In our first alternative the three interface types A-C are assumed to be identical of some type F. In practice, this approach would lead to a communication model (and a supporting language) which primarily regards communication requirements at the distributed OS level and simply forces applications to use the same model even if inadequate for the intended application scope. Most existing research projects on distributed systems take this view as pointed out in the introduction. Fig. 2 shows the resulting system architecture. It is sufficient to provide a distributed kernel with communication primitives supporting directly the functionality F. All modules of type APM or OSM will use these primitives for communication across the identical interfaces A, B and C. Although this approach greatly simplifies the design of distributed systems it might put unacceptable limitations on the distributed applications.

Alternative 2 : A ⊂ C

In this alternative it is assumed that applications are written in a language which relies on a subset of the communication mechanisms as provided for the communication between OSM's. The required architectural support is basically the same as depicted by Fig. 2 if one replaces the function set F at the kernel interface by C. The overall judgement of this alternative is the same as for alternative 1.

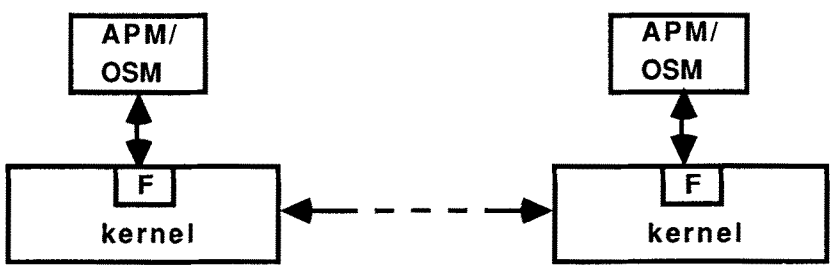

Fig. 2 Kernel architecture which supports the communication functions F as the only interface between APM and OSM modules

Alternative 3 : C ⊂ A

In this alternative the communication requirements at the distributed OS level are considered a subset of those provided for applications. The architectural support is identical with alternative 2 if the function set F is replaced by A in Fig. 2. Although both alternatives 2 and 3 are comparable with respect to the resulting system architecture there seems to be a substantial difference in practice: the communication mechanisms of alternative 3 provided by the kernel are modelled primarily with respect to the scope of the applications under consideration. OS requirements are integrated into the communication model by appropriate extensions.

It is expected that this approach will yield communication interfaces with a richer set of communication primitives than obtained by the opposite view taken by alternative 2. The successful application of this approach dictates, however, that the requirements for the interface types A and C harmonize.

As an example, let us assume that the intended applications for a distributed system are sufficiently supported by the functions SEND, RECEIVE, REPLY, COPY_TO and COPY_FROM with the semantics as defined for the V-kernel [CHE84].

At the distributed OS level the requirements for appropriate communication support might have been defined by the functions SEND, RECEIVE, REPLY and the additional demand for a multicast capability. The multicast capability can be achieved by the introduction of process groups and the additional function GET_REPLY as explained in [CHE85]. Both provisions are natural extensions of the original model and can easily be integrated into a final set of primitives represented by the functions SEND, RECEIVE, REPLY, COPY_TO, COPY_FROM and GET_REPLY.

Alternative 4 : $A \neq C$

So far we have discussed alternatives which lead to kernel architectures directly supporting the interfaces A, B and C. The notation $A \neq C$ will be used now to indicate that A and C are not subsets of each other. This is the most realistic assumption. Two different subcases can be distinguished.

Subcase 4.1 : A <-- C,

This case is characterized by the fact that C is an adequate basis for the construction of A (expressed by A <-- C). We generally consider this property being fulfilled if C is more primitive, less restrictive and less reliable than A. The required architectural support for this subcase is illustrated by Fig. 3. The kernel offers directly the functionality as required by the distributed operating system (interface C). The higher level needs of distributed applications are supported by kernel extension packages (KEP's) which are constructed out of the kernel primitives. Since A is based on C the communication between APM's and OSM's is performed by the transformation of service requests to OSM's into corresponding primitives of C. As an example it is conceivable to provide an asynchronous, notification-oriented and unreliable communication mechanism at interface C while an RPC mechanism with at-most-once semantics is provided at the interface A. It has been shown by several implementations that a reliable RPC can be sufficiently built on top of an unreliable asynchronous message passing mechanism [BIR85].

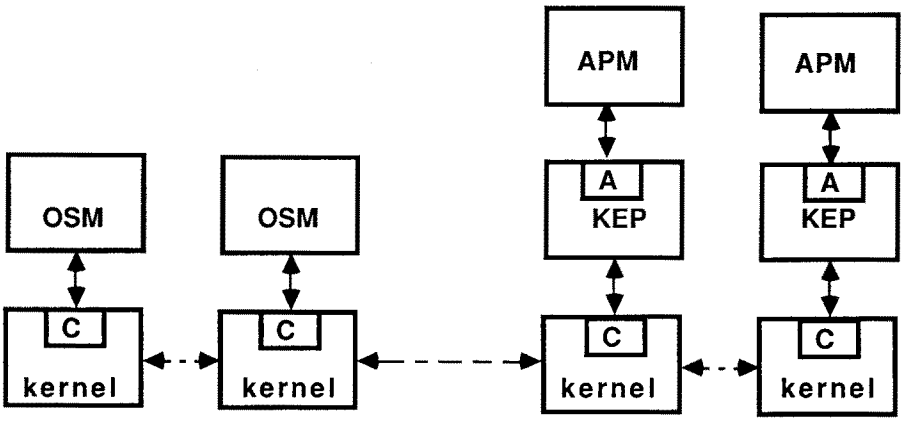

Fig. 3 System architecture for hierarchically dependent communication mechanisms at the OS and application level

Subcase 4.2 : A <#> C,
This is the most general case since no assumptions are made about direct relations between A and C (expressed by A <#> C). The only assumption we make is that it is always possible to find a common primitive communication model S for which it is true that

C <-- S
A <-- S

i.e. both interfaces C and A can be constructed out of S. The resulting system architecture is shown in Fig. 4. The distributed OS kernel offers the primitives for the communication model S which is neither sufficient for describing communication issues at the distributed OS level nor at the application level nor between them. Hence, it is mandatory to provide different KEP's for the support of OSM's and APM's on top of the kernel. In order to facilitate communication between APM's and OSM's the kernel extension packages have to provide the functional support for the interface B in addition to A or C respectively.

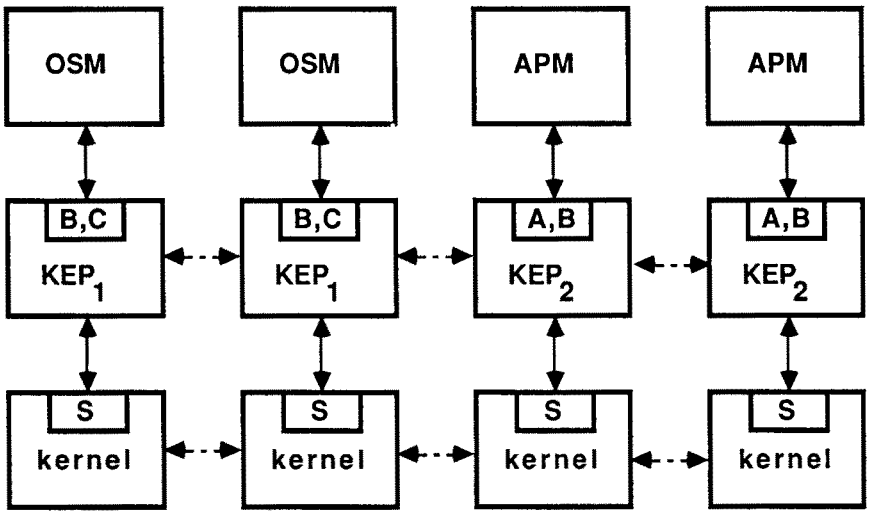

Fig. 4 System architecture for the support of different communication models at the OS and application level which are based on the common communication mechanism S

4. Multiple communication interfaces in the INCAS-project: A case study

The INCAS* multicomputer project [NEH87] was started in 1983 with strong emphasis on methodological aspects in the design of distributed systems. A topic of the project was the support of program development at the operating system and application level by powerful distributed programming languages. It is worth to notice that long before the INCAS project was started the two languages LADY (Language for Distributed Systems) and CSSA (Computing System for Societies of Agents) had been developed independently by two different research groups for different purposes.

The development of LADY as an implementation language for distributed operating systems was started in 1980 at the University of Kaiserslautern. A first prototype was operational in 1983 on a network of TI-990 microcomputers [MAS84]. The development of CSSA dates back to 1977 [BOE77] by a research team at the University of Bonn. The intended application scope was closely related to concurrent AI algorithms. In 1983 both research teams joined to form the INCAS project at

* (Incremental Architecture for Distributed Systems, funded by the Deutsche Forschungsgemeinschaft as part of the SFB 124)

the University of Kaiserslautern and decided to build an experimental distributed system supporting advanced versions of both languages for the design of distributed operating systems and application programs. The operating system team took the opinion that a sufficient design methodology for distributed operating systems should not put any constraints on the communication model on which distributed application languages are based. The implementation of a run time environment for CSSA by LADY was considered a test case for the suitability of the underlying structuring concepts in LADY.

As the next step we will give a short overview of both languages with special attention to the communication models.

The implementation strategy for both languages follows the alternative 4.1 as described in the previous section, i.e. it was assumed that the communication model in CSSA could be easily constructed out of the primitives of LADY. After having sketched the general implementation strategy we will discuss some problems encountered during the implementation phase.

4.1 Overview of the language LADY

The LADY language reflects our view of an adequate linguistic support for describing distributed operating systems. The structuring concepts of LADY are expressed in terms of three language levels as illustrated in Fig. 5 [WYB85].

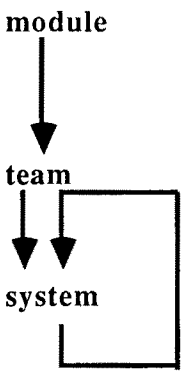

Fig. 5 Language levels of LADY

The fundamental structuring unit of LADY is the team. A team consists of a collection of tightly coupled processes which communicate via shared memory by using monitor modules or lower level synchonization primitives such as semaphores. Teams are considered the smallest indivisable distribution units, i.e. they have to be placed as a whole at one processing node. Several teams constitute a system.

Systems and teams may be combined into a higher level system. This definition of systems allows nested system structures of any depth.Teams interact with other teams via message passing. A port interface encapsulates the internal structure of teams against the external environment. Ports are typed, i.e. they can only handle messages of a given type. Message types can be defined as arbitrary structures of fixed length. The port concept in LADY is symmetric as in NIL [STR85] : input ports define the message interface exported by a team, while output ports define the message interface imported by a team from its environment. A process can send a message to a destination only if a connection between the output and a corresponding input port has been established beforehand.

Two types of connections between input and output ports can be defined:
a) logical channels, which provide for a one-to-one link between an output and an input port;
b) logical buses, which provide for a many-to-many link between output an input ports, thereby offering a multicast communication capability.

Input ports can be associated with a buffer of fixed length at declaration time which allows to store a maximum number of messages of a given type. The buffer capacity can be specified to be zero.
The semantics of one-to-one communication via logical channels can be described as follows:
a process attempting to send a message to a receiver suspends execution until the message has been successfully stored at the receiver's site (either in the buffer or in the local working store of the receiving process in case that buffer capacity zero was specified). Symmetrically a receiving process is blocked until a message has arrived at the addressed input port and copied into the receiver's local working store. (A timeout mechanism is also provided for an abnormal termination of a SEND or RECEIVE operation). If no buffer space has been associated with an input port, the one-to-one communication is semantically equivalent to the synchronization send [LIS79]. If buffers are envolved the described semantic falls into the class of asynchronous communication.

Logical buses offer three distinct transmission modes, which differ in their addressing selectivity:
a) a broadcast message which is sent to all input ports connected to this bus;
b) a multicast message which is sent to all input ports at the logical bus which belong to the same port group (port groups are defined by special port group identifiers)
c) an individual message sent to a single input port connected to the bus.
The transmission modes are selected by different SEND statements.

Reliability of varying degree can be achieved for multicast/broadcast operations by an additional function which allows to dynamically define the expected success of a SEND-operation in terms of

the number of positive acknowledgements from receivers (the default being 0). Notice that acknowledgement messages are sent automatically by the kernel if not explicitely disabled at the receiver's site.

More details on the communication model in LADY can be found in [WYB86].

4.2 Overview of the language CSSA

The underlying computational model of the language CSSA is based on the notion of actors called agents in CSSA originally developed by Hewitt [HEW77]. An agent is an active unit consisting of a cluster of operations which can be activated by receiving messages from other agents. Messages arriving at an agent while it is performing an operation are collected in a mailbox associated which each agent, i.e. messages are processed one at a time.

The message passing scheme in CSSA is asynchronous and notification-oriented. A message may be issued by the statement

SEND <op-name> <message> TO <target-agent>

which is a non-blocking operation. A multicast send is possible by specifying a set of agents as the target.

The set is specified by either a list of agent names or by an agent type. In the latter case the set is defined by all agents instantiated from this type.

A sender can request a reply by specifying
SEND . . . . . REPLY TO <op-name>

The receiving agent responds to such a reply-obligation by issuing a
REPLY <message>

at the end of an operation. The target agent and the operation name are obtained from the message header of the message being processed. CSSA distinguishes between implicit and explicit message receipt. Since these differences are irrelevant for the underlying communication model only the implicit message receipt will be discussed here (the reader is referred to [BEI85] for further details).

An agent basically consists of a set of variable declarations and several clusters of named operations:

<var-decl>
OPERATION <name> <pattern> <assertion>
    IS
    •
    •
    •
    END OPERATION
<further operation definions>;

The global variables constitute the state of an agent. The agent, when not executing an operation, scans its mailbox for executable messages. A necessary condition for a message to trigger the activation of an operation is the matching of the operation's name in the agent specification with an operation name contained in a message. The <pattern> and <assertion> part in the operation header allow to specify additional conditions for the activation of an operation by arriving messages. The definition of the <pattern> field in the operation's header enforces a pattern match between <pattern> and the message contents before the message is processed. By defining the <assertion> field the user can specify an arbitrary predicate on the values of the message parameters and variables of the agent. The message transport is assumed to be totally reliable at the CSSA-level, but message propagation delays are assumed to be of undefined and finite length (as a consequence messages may be received out of order).

4.3 Overview of the system architecture

Fig. 6 shows the overall system architecture of the INCAS-system. The distributed operating system kernel directly supports the communication model of LADY. Therefore, the kernel is only aware of teams as the communicating units.

Two different classes of teams can be distinguished.

- teams which implement a service of the distributed operating system like a printer-team,
- teams which provide the run time environment for agents. We call these teams agent-servers. There is a one-by-one correspondence between agents and agent-servers.

The principal structure of type agent-server is shown in Fig.7 (details are ignored). It consists of three processes and two monitors.

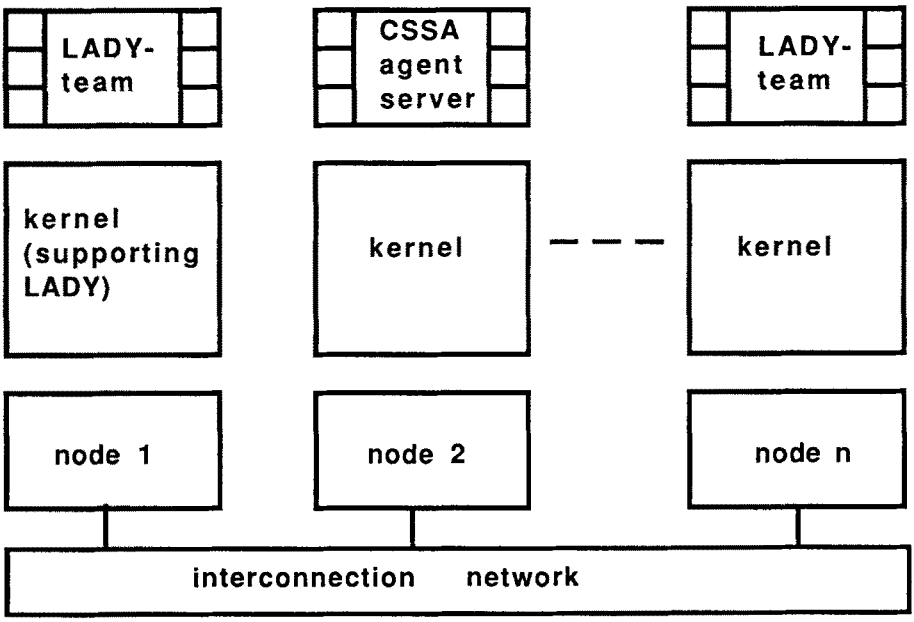

Fig. 6 INCAS - overall system architecture

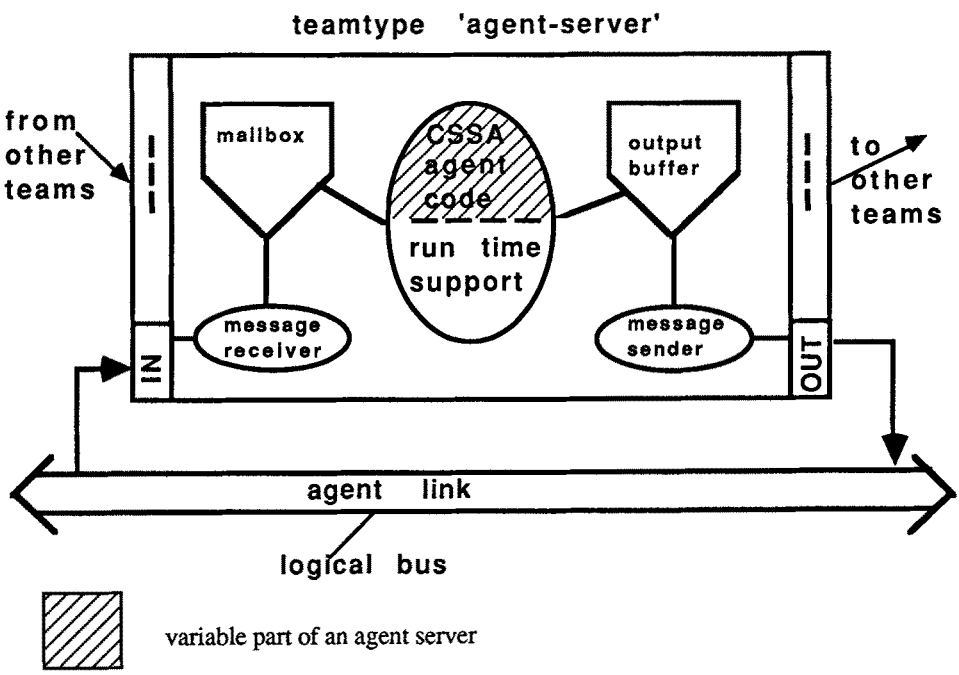

Fig. 7 Mapping of a CSSA-agent into a LADY-team

The processes 'message receiver', 'message sender' and the monitors 'mailbox' and 'output buffer' are needed to implement the asynchronous message interface for agents and to make arriving messages accessable for the pattern matching access routines of CSSA. The process 'CSSA-agent' contains the agent-code together with a run time support package which - besides other functions - transforms system call's into the appropriate sequence of communication primitives at the LADY level. The communication between agents at the CSSA level is achieved by linking together all agent servers via the logical bus 'agent link'.

5. Discussion of particular problems

The implementation scheme outlined in the previous chapter which follows closely the alternative 4, subcase 1 as discussed in chapter 3 has proven to be generally feasible. However, a deeper look into some of the problems encountered during the implementation phase shows that it is very hard in practice to assure that A <-- C holds, i.e. that the communication model of the distributed operating system is a sufficient basis for implementing the communication model at the application level. Below follows a discussion of the most serious problems encountered during the system implementation .

Problem 1: Message selection
Messages in LADY are received in FIFO order. No special selection mechanisms exist to select messages from the associated buffer of an input port. In CSSA comfortable means are provided by the <pattern> and <assertion> field of an operation for message selection. The support of this mechanism by LADY required to make all messages accessible to the pattern matching algorithms as soon as they arrive at an input port. This has been achieved by the additional process 'message receiver' and the monitor 'mailbox' and results in an additional message copy (from the process local working store to the monitor).

Problem 2 : Non-blocking SEND operation in CSSA
In order to implement a truly non-blocking SEND operation for CSSA all messages sent by an CSSA-agent are first buffered in a monitor 'output buffer'. The messages are then physically transferred by the process 'message sender'. In total, each CSSA-agent requires for its implementation three (light-weight) processes and two monitors.

Problem 3 : Incompatible multicast models
Multicast communication in CSSA is based on naming all agents in one SEND-operation either by specifying them in a list of receivers or by providing an agent type meaning that all agent objects of this type are addressed. The latter case can be directly mapped to the concept of port groups in LADY where all agents of a given type are represented by one specific port group. However, the multicast mechanisms of LADY cannot be used if the receivers are listed directly in a SEND-operation. The difficulties are caused by the different addressing schemes in both languages: In LADY, the name of a logical bus serves as a multicast group name which is used by the communication subsystem to route

the message to all the receivers, while in CSSA the set of receivers may dynamically change for each SEND-operation and so an unknown number of multicast groups can exist at run time. Even the establishment of a multicast group prior to the sending of a message would not solve the problem, since the maximum number of multicast groups a team can belong to is fixed at compile time in LADY.
Another source of incompatibility is the difference in reliability of the multicast approaches in both languages. CSSA assumes total reliability, i.e. also 'exactly once' semantics for a multicast. Even the highest possible degree of reliability of a LADY-multicast does not avoid undetected loss of messages at particular receivers.

Problem 4 : Variable length messages in CSSA
Ports in LADY are typed with fixed length messages. Messages in CSSA are unbounded and of variable structure. Since all message traffic between CSSA agents is performed through the two standard ports IN and OUT of the team type 'agent server' interconnected by a logical bus we were generally faced with the problem of mapping variable length messages at the CSSA-level into fixed length messages at the LADY-level. Two-level disassembling/assembling of messages at the LADY and CSSA level could be avoided by limiting the message size associated to agent server ports.

Other minor sources of problems in the implementation of a CSSA run time environment based on LADY are related to the dynamics in both languages. We will not discuss these issues in more detail because it is behind the scope of communication models.

6. Conclusions and lessons learned

The discussion has shown that there is a serious need in distributed systems for the support of different, possibly rather incompatible communication models due to varying communication requirements at the distributed OS and application level. In addition, different classes of distributed applications may have conflicting requirements with respect to the degree of synchrony and reliability.

If the intended scope of the distributed applications is well known at system design it might be possible to define a single universal communication interface which meets the OS requirements as well as the applications requirements. However, if this is not the case (i.e., the application scope is not well defined) special care must be taken in order to avoid performance penalties and / or loss of desired functionality.

It is recommended to provide a kernel with simple but very universal communication primitives and to assemble specific interfaces for the distributed OS and application levels out of such a kernel.
In the INCAS project we could solve most of the problems encountered during system implementation by an iterative design step which involved also some minor changs to the LADY

language. The time consuming copying process for messages in and out of the monitors 'mailbox' and 'output buffer' within CSSA-agent servers could be finally circumvented by introducing the concept of 'shared storage' directly accessible to tightly coupled processes within a team. However, we did not find an acceptable way to bridge the different reliability views taken in the multicast models of the languages LADY and CSSA.

Acknowledgement

I would like to acknowledge the valuable discussions with my staff members F. Mattern and D. Wybranietz on early drafts of the paper.

References

[MIL87] B.P. Miller, D.L. Presotto, M.L. Powell: DEMOS/MP: The Development of a Distributed Operating System, Software-Practice and Experience Vol. 17, No. 5, 277-290 (1987)

[WAL83] B. Walker, G. Popek, R. English, C. Kline, G. Thiel: The LOCUS Distributed Operating System, Proc. 9th SOSP, 49-70 (1983)

[MUL84] S.J. Mullender, A.S. Tanenbaum: The Design of a Capability-Based Distributed Operating System, Report CS-R8418, Amsterdam, The Netherlands, 1984

[CHE84] D. Cheriton: The V-Kernel - A Software Base for Distributed Systems, IEEE-Software Vol. 1, No. 2, 19-42 (1984)

[BLA85] A.P. Black: Supporting Distributed Applications: Experience with Eden, Proc. 10th SOSP, 181-193 (1985)

[SWI85] D.C. Swinehart, P.T. Zellweger, R.B. Hagman: The Structure of Cedar, Proc. of the SIGPLAN 85 Symposium on Language Issues in Programming Environments, 230-244 (1985)

[LIS83] B. Liskov, R. Scheifler: Guardians and Actions: Linguistic Support for Robust Distributed Programs, ACM TOPLAS VOL. 5, No. 3, 381-404 (1983)

[AND82] G.R. Andrews: The Distributed Programming Language SR-Mechanisms, Design and Implementation, Software-Practice and Experience 12, 719-753 (1982)

[SLM85] G.T. Almes, A.P. Black, E.D. Lazowska, J.D. Noe: The Eden System: A Technical Review, IEEE Trans. on Software Engineering Vol. 11, 43-59 (1985)

[STR85] R.E. Strom, S. Yemini: The NIL Distributed Systems Programming Language: A Status Report, ACM SIGPLAN Notices Vol. 20, No. 5, 36-44 (1985)

[CAR86] N. Carriero, D. Gelernter: The S/Net's Linda Kernel, ACM TOCS Vol. 4, No. 2, 110-129 (1986)

[SCO87] M.L. Scott: Language Support for Loosely Coupled Distributed Programs, IEEE-Trans. on Software Engineering Vol. 13, No. 1, 88-103 (1987)

[SCO86] M.L. Scott: The Interface Between Distributed Operating System and High-Level Programming Language, Proc. of the Int. Conference on Parallel Processing, 242-249 (1986)

[RAS81] R. Rashid, G. Robertson: Accent: A Communication Oriented Network Operating System Kernel, Proc. of the 8th SOSP, 64-75 (1981)

[FIT86] R. Fitzgerald, R.F. Rashid: The Integration of Virtual Memory Management and Interprocess Communication in Accent, ACM TOCS Vol. 4, No. 2, 147-177 (1986)

[LIS85] B. Liskov: Limitation of Synchronous Communication with Static Process Structure in Languages for Distributed Computing, Techn. Report No. CMU-CS-85-168, Carnegie-Mellon-University, 1985

[JUL85] E. Jul: A Classification of Distributed Operating Systems, Techn. Rep. 85-05-01, Dept. of Computer Science, Univ. of Washington, 1985

[SHA84] S.M. Shatz: Communication Mechanisms for Programming Distributed Systems, IEEE-Computer Vol. 17, No. 6, 21-29 (1984)

[NEH87] J. Nehmer, D. Haban, F. Mattern, D. Wybranietz, D. Rombach: Key Concepts of the INCAS Multicomputer Project, IEEE Trans. on Software Engineering Vol. 13, No. 8, 913-923 (1987)

[BIR85] A.D. Birell: Secure Communication Using Remote Procedure Calls, ACM TOCS Vol. 3, No. 1, 1-14 (1985)

[CHE85] D.R. Cheriton, W. Zwaenepol: Distributed Process Groups in the V-Kernel, ACM TOCS Vol. 3, No. 2, 77-107 (1985)

[MAS84] R. Massar: LADY - Design and Implementation of a Language for Distributed Systems, Ph.D. - Thesis, University of Kaiserslautern, Dept. of Computer Science (1984)

[BOE77] H.P. Böhm, H.L. Fischer, P. Raulefs: CSSA-Language Concepts and Programming Methodology, SIGPLAN -Notices, Vol. 12, No. 8, 100-108 (1977)

[WYB86] D. Wybranietz, D. Haban, P. Buhler: Some Extensions of the Language LADY, Techn. Report SFB 124-28/86, Dept. of Computer Science, University of Kaiserslautern 1986

[LIS79] B. Liskov: Primitives for Distributed Computing, Proc. of the 7th SOSP, 33-42 (1979)

[HEW77] C. Hewitt: Viewing Control Structures as Patterns of Passing Messages, Artificial Intell. Vol. 8, 323-364 (1977)

[BEI85] C. Beilken, F. Mattern: The Distributed Programming Language CSSA - A Very Short Introduction Dept. of Computer Science, Univ. of Kaiserslautern, Techn. Report 123/85 (1985)