Group Orientation: a Paradigm for Modern Distributed Systems

Paulo Veríssimo
paulov@inesc.pt
Tech. Univ. of Lisboa
IST - INESC [*]

Luís Rodrigues
ler@inesc.pt
Tech. Univ. of Lisboa
IST - INESC

Werner Vogels
werner@inesc.pt
INESC

Abstract

Increasing use of distributed systems, with the corresponding decentralisation, stimulates the need for structuring activities around groups of participants, for reasons of consistency, user-friendliness, performance and dependability. Although there is a significant number of group communication protocols in the literature, they are penetrating too slowly in operating systems technology. Two important reasons are: the literal interpretation generally made of the end-to-end argument, and the lack of a layer mapping end-user needs (management of replication, competition, cooperation and group membership) into what is generally provided by the communication layer: agreement and order properties.

The paper discusses both problems, proposing ways for structuring systems and defining building blocks for group-oriented activity, using concepts like object groups. It suggests that the group concept should pervade the whole architecture, from network multicasting, to group communications and management. Emerging technology will help materialise these concepts.

Introduction

The increasing use of distributed systems is in part due to the requirements of inherently decentralised activities, such as computer supported collaborative working, or distributed computer control. It becomes then natural for a number of such computations to be structured around groups of participants, for reasons of consistency, user-friendliness, performance and dependability.

When looking at the several classes of distributed activities that may take place in a distributed system, the group concept appears intuitively: when a group of participants cooperate in an activity (e.g. management of a fragmented database, distributed document processing or distributed process control), compete in an activity (e.g. to share a given resource), or execute a replicated activity for performance or fault-tolerance reasons (e.g. replicated database server, replicated actuator).

Group-orientation, as a general rule, does not imply group-oriented programming for the end user. That is, group tools can help the programmer build applications where groups are *visible*, because they are part of the problem specification. However, they may also be useful to implement system support functions used transparently by the programmer, that is, in an *invisible* way. In fact, the utility of groups pervades all layers of a distributed architecture, from multicasting network infrastructures, through group communication and membership, to the group activity management services. They may be at the core of efficient implementations at several levels of abstraction: operating system (e.g. microkernel port group management), internetworking (e.g. routing and state dissemination and management of groups of routers), distribution support (e.g. replication management, distributed locks), applications (e.g. cooperative editors, teleconferencing, manufacturing cells). In some applications, the end user will not see groups at all, reasoning in terms of object invocations, RPCs, and so forth. However, underlying that users's programming environment, groups may be fulfilling their rôle in transparently supporting a distributed shared memory mechanism, a replicated object mapper, a distributed lock manager, etc.

Regardless of the current preponderance of other paradigms, other than group-orientation, for programming distributed applications, a few observations are

[*]Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 - 6^o - 1000 Lisboa - Portugal, Tel.+351-1-3100281. This work has been supported in part by the CEC, through Esprit Projects BROADCAST and DINAS, and JNICT, through Programs Ciência and STRIDE. This paper is an extended and revised version of a paper that appeared in Procs of the ACM SIGOPS 1992 Workshop, Mont Saint Michel-France.

in order:

- Real-time – or *responsive* – systems deal with the environment, thus handling of events is inevitable. In event-driven systems, events "travel" inside the (distributed) system in the form of messages, or message-interrupts, before being transformed. There is a number of applications being better addressed in the domain of events [40];
- in highly concurrent and/or interactive systems (e.g. collaborative group work or distributed computer control) message-passing paradigms are necessary;
- remote procedure call, being blocking, unilateral and asymmetric (client-to-server), has some shortcomings, more evident in the kind of systems just mentioned; it should be complemented with paradigms supporting multilateral, non-blocking and peer-to-peer interactions [1].

These observations lead us to the conclusion that a distributed computing platform can only benefit from the coexistence of remote-operation based paradigms, such as RPCs, with message-passing, diffusion-based ones, such as group protocols. This, without denying the validity and usefulness of either one.

The requirements of highly distributed activities are not adequately satisfied by the basic interfaces traditionally supplied as "distribution support", such as "sockets" or "streams": these are semantically too poor, and most of them are not multi-participant. Building blocks for group activity have been studied in the past in pioneering projects such as the V-kernel [11], ISIS at Cornell [5], the Circus project [12], and also [1,19]. They are currently the subject of great interest, illustrated by projects as the PSYNC/x-Kernel work at Univ. of Arizona [29], the work on object groups by ANSA [3], the work of Molina [27], the IBM flight control AAS [17], the European DELTA-4 project [30], the work about groups in Arjuna [23], the current work on Isis/Horus [31].

In order for applications with high levels of concurrency to be correctly designed, have acceptable performance, and remain operational for long enough, whatever distribution support environment to be conceived must combine: encapsulation, modularity and diversity; fault tolerance and timeliness; distributed algorithmics; support for actions and data (events and state).

[1]Engineering-wise, some of these problems have of course been solved long ago one way or the other (replicated RPC, "asynchronous" RPC or RSR, calling process fork, etc.). However, they should be properly addressed at the model level, if possible.

The combined notion of *object* and *group* may provide designers and programmers with a suitable framework to achieve those objectives. Practically all works cited above have addressed a part of the problem, and a few of them have tried to systematise solutions.

In this paper, we discuss how to structure groups in distributed systems, from networking to application support. We start by dissecting two main arguments against using groups: the end-to-end argument and group visibility. We continue by presenting an overall picture of the necessary building blocks of a group-oriented system, then detailing each of the blocks. Finally, we do a discussion about programming over a group-oriented system, either with group visibility or not.

The arguments against groups

An existing reluctance to having services with high quality (e.g. reliable multicast) provided off-the-shelf in a system, has been related to the literal interpretation often made of the *end-to-end argument* [35]. In fact, the argument is against providing more functionality than needed at a given layer of a system, because this goes against optimising efficiency, risks introducing redundant protocol actions like error recovery steps, and ultimately, goes against good sense. Nevertheless, if a class of applications requires a certain functionality (or quality of service), however complex it may be, the lower layer or more generally the *operating system support*, should provide it. It frees the user from programming it, and will probably have been optimised and widely tested when supplied with a system.

In fact, a number of classes of distributed activities can be defined whose requirements are solved by a set of distributed algorithms. In that case, supplying a suite of the corresponding protocols is in essence a correct interpretation of the end-to-end argument, if done in a way such that the users not requiring them are not penalised by their existence. This reasoning applies to all levels of the architecture, from network hardware through communications to computing.

One such "semantically-loaded" class of protocols is that of group communication. Though there is a significant body of research in this area, they have had a slow penetration in operating system technology. We are convinced that there is a major reason for this fact, beyond the end-to-end argument one: *visibility of the group communication protocols* is often awkward for the applications programmer. It is nec-

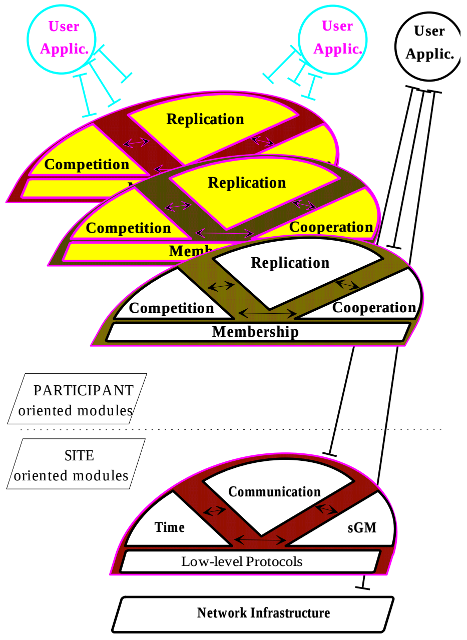

Figure 1: Group Support Building Blocks

essary for the services mentioned above to be systematised and mapped into the programmer's universe. A convincing suggestion is hiding them *under* an RPC environment. This solves only part of the problem, namely when one wishes to replicate, maintaining a client-server relationship. It does not contribute greatly when that relation does not apply or is *contra natura* – we have cited some examples in the introduction. For these cases, paradigms which adjust to the event-driven and peer-to-peer nature of autonomous processes apply. They should however be encapsulated in a way as to avoid the inconvenients of directly programming with message-passing.

We believe what is missing in a group-oriented system structure, is what might be called in group jargon, a *group activity management* layer. In essence, a building block that controls how groups of participants interact, in a number of well-defined classes of distributed activities. This layer should address problems such as: concurrency control, mutual exclusion, distributed parallel processing, replicated processing, replicated data, etc. Though there are a large number of specific solutions for these problems published, some of them group-based, a systematics for these services is not clear yet. We risk saying that it should consist of a combination of protocols for controlling basic group activities: membership, replication, sharing and cooperation.

Structuring Group Support in Distributed Systems

The necessary building blocks for group-oriented system structuring and programming are represented in figure 1. The notion of group pervades all layers of a distributed architecture, from multicasting communication infrastructures, group communication and time services, to the group activity management services just mentioned.

Let us call *participant* to the end-user of the system support. On the other hand, let us call *site* to the machine as seen from the perspective of the network: no matter how many participants it hosts, messages directed to them are funneled through this single attachment, the machine's network adapter. In consequence, given that group tools are fundamentally protocols, it is useful to structure them depending on among what they run: inter-participant protocols run among *processes*, inter-site protocols run among *processors* (even if on behalf of processes) [38]. Figure 1 makes explicit which are the site- and the participant-level modules.

Network Infrastructure

A number of distributed protocols in the recent years have been designed to be network-independent [21,5,27]. However, in trying to be generic and scalable, they do not take advantage of the existence and the emergence of network technologies such as LANs and MANs. In consequence, while these will perform well either in local or wide areas, synchronous or asynchronous environments, etc., users will be less and less prepared to pay the cost of technology independence when technology is there. Most local enterprise, institution or factory settings run over LANs. The near future will see organisation-wide distributed systems over metropolitan-area networks. These are reliable both in terms of error rate and availability, display from high to very high speed and bandwidth, have broadcasting/multicasting facilities, and some are capable of real-time operation and have reasonably low delivery delays. More remotely, meshes of ATM switches will provide very high interconnectivity and speed down to the workstation, in a fairly large geographical scope.

We suggest that protocols be prepared to take the best advantage of the technology they have available. In the model we follow, there are a few key rules: abstract from particular networks but recognise a few *network classes* (ex. LAN, INTERNET, MAN, ATM, etc.); define a set of properties for each class; admit different abstract networks at different layers of the infrastructure[2]; run parts of the protocol at different abstract network layers, by *delegation* [?].

For the sake of example, a reliable multicast protocol running over a large-scale network like the Internet, would begin its execution by seeing an INTERNET class network with its weak properties, but might delegate part of its execution on protocols running local to the destination LANs, seeing a LAN-type abstract network, and obviously taking advantage of that.

We find this is a satisfactory compromise for the long-lasting contradiction between the low-level approach (not-scalable) and the high-level one (not efficient). We may now present a set of requirements of the network infrastructure in order to efficiently support groups. In view of the above, some of these features may be provided by only a part of the infrastructure, but still play their rôle in group communications efficiency:

- *logical group addressing* for recipient number and location transparency;
- hardware *multicast* and *selective* addressing for group and sub-group addressing;
- *address resolution:* handling a large number of addresses efficiently;
- *topology* should ease multipoint communication;
- *reliability* and *availability* become more important: there are more players involved than in traditional point-to-point.

Group Communication

The modules in the communication support system are concerned with site-level protocols. It may sometimes be relevant to include a sub-layer of *low-level protocols* which the group communication protocols may rely on. This occurs for example as a set of "super-services" of the network infrastructure which one does not wish to include in the latter. As an example, one may have a basic service of efficient *large-scale multi-destination dissemination*, with *varying reliability.*

Group communications services rely on the low-level network services. They ensure that a group of participants exchange messages following a set of rules, without worrying about how they are secured. The semantics of group communications services can be characterised by combinations of agreement, order and synchronism properties. These elementary properties have been characterised in the literature in the recent years. Surveys can be found in [8,30,28].

[2]There is an analogy with the ISO layering here, though using different criteria.

For example, the strongest form of agreement is unanimity, where any message delivered to a recipient, is delivered to all correct recipients. Unanimity may be unnecessary in some situations. For instance, queries to a replica group need only reach one of replicas, or a quorum of them, it does not matter exactly which. Relaxed forms of agreement apply then, like ensuring delivery to a number of recipients N ($N = 0$ is the well-known datagram semantics).

In a distributed system, participants must perceive the order in which actions and events take place. The cause-effect relation is the natural ordering of events in a system. It is called a causal order. This order may be relaxed in some cases, for example to a FIFO (first-in-first-out) order, if senders are not causally related. For example, when requests from different clients to a server are commutative. On the other hand, if a participant is actively replicated, messages to the replicas should be sent in the same order. This is called a total order.

Synchronism of a group protocol can be measured by its *steadiness*, the greatest difference between delivery times observed at one site, and its *tightness*, the greatest difference between delivery times observed in one execution. According to this, there is a spectrum from tightly-synchronous [14], through loosely-synchronous [37], to asynchronous protocols [29,5], depending on whether those differences are large or small, compared to the execution time, or even not bounded at all. From the degree of synchronism depend not only real-time but also ordering capabilities [40].

In conclusion, a group communication subsystem should:

- be prepared to accommodate grades of *agreement, order and synchronism*;
- supply a *range of services* each formed by a combination of some of the properties above;
- select those services to fulfill user needs, i.e. making a *good use of the end-to-end argument.*

Rationale for this exercise can be found in [21,5,30, 26].

Site membership

The problem of membership is the problem of knowing who or what belongs to a system or group, or is present in an activity. At this point, it is important to draw a distinction not taken into account by most group-oriented systems we know of: the difference between the site and participant membership problems.

In a distributed application, there are several participants, sometimes more than one at each site. Having rules to assess, and sometimes control, their arrival or departure, is most of the times useful if not mandatory [32]. This kind of activity is *participant membership* management. However, for each set of participants engaged in a distributed activity, there is another important set: the set of sites hosting those participants, smaller than or equal to the participant set. The management of this set has different requirements from the management of participants, so call it *site* membership management [34].

We believe it is of extreme importance to recognise these membership levels, so to speak. Firstly, it may simplify the construction of protocols and of the participant group management protocols. As an example, consider a site with 500 groups. When it fails, a system without 2-level membership will trigger 500 executions of the group membership protocol in parallel. In a system with 2-level membership, the site membership protocol may handle the site failure and intelligently propagate that information to the participant membership protocols.

Additionally, it separates processor and communication (site) failure detection from process (participant) failure detection. While improving the accuracy and fairness of such detection, this separation has an important consequence in asynchronous systems. For these systems there is a well-known result stating that it is impossible to guarantee consensus [18], exactly due to the impossibility of telling a slow participant from a failed one. Theoretically, by separating site and participant failure detection, one narrows the domain under the reach of the FLP result. Site failure detection remains unreliable, whereas participant failure detection, performed locally, can be made reliable.

Incidentally, in the field of global-clock-less, acknowledgement-based protocols [9,4,29,37], the site and participant membership functions have often been aggregated. This separation makes it clear that the requirements of these protocols, to do correct inter-site communication, pertain to site membership management (e.g. if the protocol uses acknowledges, it is necessary to have a coherent view of the group of sites from which replies are expected).

Time and Timing

The importance of time in distributed systems has been largely underestimated. Real-time systems require the ability to control duration of activities, response time, etc. A global reliable time-base accessible by all nodes is thus mandatory for distributed real-time systems. However, since systems are becoming more interactive (multimedia, CSCW, etc.), real-time tends to be a necessary attribute of systems in general.

On the other hand, a number of distributed algorithms are based on the existence of a global time notion. Even non-real-time or soft real-time applications can benefit from these algorithms and thus we claim that a reliable, precise and accurate time-base is a very useful building block in any distributed system. With regard to the way "time" is shipped in most existing distributed operating systems, this involves more attention to:

- *precise* clock synchronisation algorithms, i.e. maintaining clocks together;
- *fault-tolerance*, i.e. maintaining the time correct, reliable and available;
- external synchronisation [3], i.e. *accuracy* vis-a-vis absolute time references (TAI, GMT, UTC).

Group Activity Management

Whilst group communication is concerned with allowing participants of a group to exchange messages and establish rules for that exchange, *group activity* management is concerned with defining and controlling the group objectives.

Distributed activities can be reduced to combinations of three fundamental operations, sharing, replication, and cooperation[4]. So in the making of a distributed application we can conceptually visualise protocols falling in one or more of these activity classes, complemented with membership management:

- *sharing* is the activity concerned with a number of participants commonly accessing an entity or resource;

[3]GPS systems are decreasing impressively in price. The time is coming when it will be feasible to have at least one GPS receptor per system, injecting highly accurate absolute time which is then distributed via internal clock synchronisation algorithms.

[4]With slightly different designations, the three-fold characterisation of distributed activities was established by LeLann [22].

- *replication* is the activity where a number of participants execute replicas of the same action, or hold replicas of the same data;

- *cooperation* is the activity where a number of participants execute fragments of an action, or hold fragments of a data entity;

An example of cooperation management is a protocol to control a task to be performed in parallel by a group of processors, or a protocol to control the simultaneous editing of a document in CSCW [2]. An example of replication management are protocols to control a set of replicated processes, in order that they perform fault-tolerant computations, ensuring whatever actions needed, like voting, collating, etc. [30].

Participant membership (we discussed site membership in a previous section) has been addressed in different contexts [15,32,25,36]. In the measure that distributed applications are decentralised, dynamic, unreliable, reconfigurable, and sometimes *large-scale*, membership assumes an essential rôle in distributed systems. Membership protocols know who is in and who is out, and control joins and leaves according to predefined rules. For example, detecting failure and re-establish the level of replication of a group, ensuring that the necessary "skills" for a cooperative activity are present in a group, preventing partitioned data divergence by allowing progress only in the majority partition.

Take the example of a fragmented and replicated database: (i) there should be membership management to control whether all the necessary modules are present — the parts of each replica, the necessary number of replicas, the majority criterion; (ii) replication management will assist in maintaining consistency of the replica set (e.g. active or passive replication); (iii) since the database is fragmented among several sites, management of cooperation between the servers controlling the parts of each replica is necessary; (iv) sharing of access to the database is handled by concurrency control, which may be implemented in a decentralised way via atomic or causal multicast protocols.

System Architecture Issues

Looking at figure 1, one may wonder how do those blocks map into a real architecture. There is no general solution, but it is intuitive that such an organisation is highly related with some versatility in programming over and within the operating system. In consequence, it might be useful to see attributes such as:

- layer compaction, layer transparency or even layer-lessness;

- interface recursivity (up- and down-calls);

- encapsulation using "large" objects;

- operating system support providing threads, efficient IPC, timer and buffer management, fast user-network information path, user-definable scheduling, easy embedding of external protocols [5].

Two alternative models for operating system support are: (a) top-level opaque interface (e.g. Unix) provided by a monolithic operating system; (b) layer-transparent interface (one of top interfaces possibly Unix), provided by several layers and modules on top of and alongside a micro-kernel, allowing visibility of the inner-most layers from user space, and the addition of extra functionality like the group support protocols. The second is clearly the preferred environment for a group-oriented architecture. It is based on the micro-kernel approach, receiving large support lately.

In this paper we go as far as suggesting, for the main building blocks (group management, group communications + time, network infrastructure):

- they should have a well-defined encapsulation and interface;

- inside these blocks a closer interaction between protocols may be desirable for performance reasons;

- they should be visible to each other and to any user program at any level, up to the top-level users.

As an example, certain clock synchronisation protocols rely on clock-less reliable group communication. On the other hand, certain reliable group communication protocols (such as the Δ-type protocols [14]) are clock-driven: they rely on the existence of the global clock provided by the time service and make heavy use of it. The group communications and time service should thus share context. The time service should be implemented at low level, as close to the network as possible. This reduces the errors in clock synchronisation and in time-stamping.

Another example concerns the group management layer. Group membership, replication, cooperation and sharing are broad designations which illustrate the basic activities concerned with supporting any flavour

[5]Network requirements have been discussed in a previous section.

of distributed computation, in combination with, and making use of, group communication. However, practical algorithms and protocols will often combine processing steps having to do with more than one of these actions. In the example already given, of the controlling protocols of a fragmented and replicated database, the user (the application builder) will probably see a "Data Partitioning/Replication ToolBox", where the elementary membership, cooperation and replication management protocols will make use of each other to perform the desired functionality. In consequence, they should share context.

As a matter of fact, nothing prevents group management and group communications from being in the same process. This may ultimately improve efficiency of their mutual interface, but they should remain as separate entities. Programming these facilities as kernel extensions is largely advisable.

Group-Oriented Programming

We will now discuss how groups can assist programming, using an object-oriented framework for reference. As noted in the beginning, groups may be perceived in two fundamental ways by users. One of them is as support for genuinely group-oriented activities, that is, ones where groups appear in the problem specification, such as in collaborative applications, where many often it is required that the participants have *collaboration awareness*. In that sense, the way to set-up the model in figure 1, is as depicted in figure 2a: user objects dialog between themselves through a public object interface which is much of a "group agent" or "group manager". The user objects merely contain their specific functionality, and they use the group manager to handle all the aspects related with their work as a group. Distribution is, so to speak, direct [7,41]. The other interaction style concerns problems where groups, not being part of the specification, may be part of the solution. That is, they may materialise the support for coordinating the sharing of an object, managing a replicated or fragmented object, and so forth. For clients of such a support environment, groups are indeed transparent [33,24,23]. This is depicted in figure 2b. Note that objects in this model have their own methods, plus the methods to enforce the group management policies.

It is perhaps relevant to specify a little better what we mean by genuinely group-oriented programming. One may find examples of its relevance in two very diverse application fields: computer supported collab-

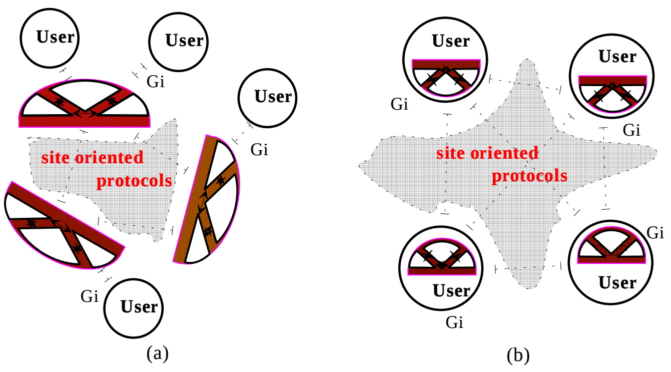

Figure 2: (a) group visibility; (b) group invisibility

orative (group) work (CSCW); distributed (computer) control systems (DCS).

Rules and means for collaborative group working are a necessary requirement of practically any collective activity, more so when assisted by the efficiency and cost-effectiveness of computer support. CSCW has been a discipline of growing interest, in the measure where widespread, ever-increasing use of communications and distributed systems make possible its application in a large number of activities, both in local and geographically broad areas.

Applications for collaborative work introduce particular requirements on the underlying system. A major requirement for the progress of CSCW is the existence of fundamental supporting mechanisms, like replicated data types, distributed locks with varying semantics, group negotiation methods, which must be merged both with user interface concepts like multiple views and dialogue encapsulation, and with the adequate architectural support (e.g. reliable and real-time group communication). Group-orientation is expected to provide a unifying approach to the task of constructing cooperative applications in a distributed environment [13].

Distributed computer control systems (DCS) are a very challenging and fast evolving field. The target systems encountered in the process control area are an ideal field to explore the notions of direct distribution, concurrency and groups. However, performance requirements and the importance attached to these problems, mostly money-critical, if not life-critical, deter the fast introduction of new concepts.

Distribution in computerised control has so far been almost exclusively limited to networking facilities, to down-load and up-load information (e.g. shop-floor data or CNC control programs), or to replace point-to-point cabling (e.g. centralised polling field-buses).

Decentralised approaches where some node autonomy is conferred are normally specialised application-level solutions.

For an evolution to take place in DCSs, such new generation distributed computer control systems must be able to provide assurances about: correctness; dependability and real-time behaviour; performance; testability. Recent architectural work (MARS [20], AAS [16], DELTA4-XPA [41]) has shown some paths to the combination of distribution, fault-tolerance and real-time, in what one could call *responsive* systems. The notion of groups is of paramount importance in these settings, given the natural division of these (sometimes large) systems in small replicated or cooperative modules, with highly concurrent operation. Groups obviate the need to reason in terms of the global system when making and proving assertions about correctness, dependability and timeliness. Group tools assist the programmer to reason about correctness of the interactions among group members.

Conclusions

The concepts advanced in this paper need not be taken as whole, or in rupture with traditional paradigms. Firstly, they are valid at different levels of abstraction: the user depicted in figure 1 may either be a programmer or an operating system nucleus. Secondly, there is a migration path which starts in the lower layers of the system. A lot may be done in improving the way distributed systems work at low level, while preserving the traditional shared-memory and/or RPC-based models to the applications programmer. Micro-kernels can take advantage of the group networking support mechanisms mentioned, or of low-level site membership management (e.g. for managing port sets). Low-level group communication protocols may assist in implementing distributed shared-memory approximations, or in performing consistent table updates.

Though the rôle of low-level group mechanisms to support high-level group-oriented programming is probably not controversial, we argue that a system structured in this way may be as effective in supporting group-oriented programs as in supporting traditional ones.

Still, a lot has to be done in what regards implementation of these concepts. A recent controversy [10,6] shows that there are no general solutions. Implementations conceived to support group-oriented programming may present shortcomings when evaluated as support of alternative paradigms, such shared memory or transactions. Still, this does not invalidate the concept, and in our opinion the solution may lie in taking problem-oriented approaches that prove to give correct and satisfactory solutions to those alternative paradigms.

References

[1] Mustaque Ahamad and Arthur J. Bernstein. Multicast communication in Unix 4.2bsd. In *Proceedings of the The 5th Intern. Confer. on Distributed Comp. Systems*, Denver, USA, May 1985. IEEE.

[2] P. Antunes, N. Guimarães, and R. Nunes. Extending the User Interface to the Multiuser Environment. In *Proceedings of the European Conf. on CSCW, CSCW Developers Workshop*, Amsterdam, September 1991.

[3] Architecture Projects Management, Ltd, Cambridge, UK. *The ANSA Reference Manual*, release 1.1 edition, July 1989.

[4] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM, Transactions on Computer Systems*, 5(1), February 1987.

[5] Kenneth P. Birman. The process group approach to reliable distributed computing. Technical report, Cornell University, Ithaca, USA, July 1991.

[6] Kenneth P. Birman. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. Technical report, Cornell University, October 1993.

[7] K.P. Birman and T.A. Joseph. Exploiting replication in distributed systems. In Sape Mullender, editor, *Distributed Systems*, pages 319–366. ACM Press Frontier Series, 1989.

[8] K.P. Birman and T.A. Joseph. Reliable broadcast protocols. In Sape Mullender, editor, *Distributed Systems*. ACM Press Frontier Series, 1989.

[9] J. Chang and N. Maxemchuck. Reliable broadcast protocols. *ACM, Transactions on Computer Systems*, 2(3), August 1984.

[10] D. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proceedings of the 14th Symposium on Operating Systems Principles*, Asheville, NC, USA, December 1993.

[11] D. Cheriton and W. Zwaenepoel. Distributed process groups in the V-kernel. *ACM Tran. on Computer Systems*, 3(2), May 1985.

[12] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, Berkeley, California 94720, USA, November 1985. ACM.

[13] François Cosquer and Paulo Veríssimo. Groupware platform definition *(in preparation)*. Technical Report RT-63/93, INESC, September 1993.

[14] F. Cristian, Aghili. H., R. Strong, and D. Dolev. Atomic Broadcast: From simple message diffusion to Byzantine Agreement. In *Digest of Papers, The 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor-USA, June 1985. IEEE.

[15] Flaviu Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Digest of Papers, The 18th International Symposium on Fault-Tolerant Computing*, Tokyo - Japan, June 1988. IEEE.

[16] Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. Technical report, IBM Almaden Research Center, San Jose,California, USA, 1990.

[17] Flaviu Cristian, Robert D. Dancey, and Jon Dehn. Fault-tolerance in the Advanced Automation System. In *Digest of Papers, The 20th International Symposium on Fault-Tolerant Computing*, Newcastle-UK, June 1990. IEEE.

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[19] Larry Hughes. A multicast interface for Unix 4.3. *Software Practice and Experience*, 18(1):15–27, January 1988.

[20] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, pages 25–41, February 1989.

[21] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Workshop on the Management of Replicated Data*, pages 31–34, Houston - USA, November 1990. IEEE.

[22] G. Le Lann. An analysis of different approaches to distributed computing. In *Proceedings of the 1st International Conference on Distributed Computing Systems*, Alabama-USA, 1979. IEEE.

[23] M.C. Little, D.L. McCue, and S.K. Shrivastava. Maintaining information about persistent replicated objects in a distributed system. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 491–498, Pittsburg, Pennsylvania, USA, May 1992. IEEE.

[24] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In *Advances in Distributed Systems*. IEEE, 1992.

[25] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. A membership protocol based on partial order. In *Proceedings of the 2nd Intl. Working Conf. on Dependable Computing for Critical Applications*, pages 1–18, Tucson, AZ 85721, USA, February 1991. IFIP WG10.4.

[26] Shrivakanth Mishra and Richard D. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report TR 92-19, The University of Arizona, Departement of Computer Science, Tucson, Arizona, USA, 1992.

[27] H. Garcia Molina and Annemarie Spauster. Message ordering in a multicast environment. In *Proceedings of the 9th Internacional Conference on Distributed Computing Systems*, pages 354–361. IEEE, June 1989.

[28] S.J. Mullender, editor. *Distributed Systems, 2nd Edition*. ACM-Press. Addison-Wesley, 1993.

[29] Larry L. Peterson, Nick C. Buchholdz, and Richard D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3), August 1989.

[30] D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer Verlag, November 1991.

[31] R. van Renesse, Ken Birman, Robert Cooper, Bradford Glade, and Patrick Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Architectures*, pages 269–283, Seattle, Washington, April 1992.

[32] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environemnts. Technical Report TR 91-1188, Cornell University, Department of Computer Science, 1991.

[33] L. Rodrigues and P. Veríssimo. Replicated object management using group technology. In *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, pages 54–61, Lisboa, Portugal, September 1993. Also as INESC AR/28-93.

[34] L. Rodrigues, P. Veríssimo, and J. Rufino. A low-level processor group membership protocol for LANs. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 541–550, Pittsburgh, Pennsylvania, USA, May 1993. Also as INESC AR/30-93.

[35] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[36] Andre Schiper and Aleta Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing*, pages 534–543, Toulouse, France, June 1993. IEEE.

[37] P. Veríssimo and José A. Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, Huntsville, Alabama-USA, October 1990. IEEE. Also as INESC AR/24-90.

[38] P. Veríssimo and W. Vogels. The changing face of technology in distributed systems. In *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, pages 119–127, Lisboa, Portugal, September 1993. also as INESC AR/15-94.

[39] Paulo Veríssimo. Real-time Data Management with Clockless Reliable Broadcast Protocols. In *Proceedings of the Workshop on the Management of Replicated Data*, Houston, Texas-USA, November 1990. IEEE. also as INESC AR/25-90.

[40] Paulo Veríssimo. Real-time Communication. In S.J. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press, pages 447–490. Addison-Wesley, 1993.

[41] Paulo Veríssimo, P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton. The Extra Performance Architecture (XPA). In D. Powell, editor, *Delta-4 - A Generic Architecture for Dependable Distributed Computing*, ESPRIT Research Reports, pages 267–294. Springer Verlag, November 1991.

[42] Werner Vogels and Paulo Veríssimo. Supporting process groups in internetworks with lightweight reliable multicast protocols. In *Proceedings of the ERCIM Workshop on Distributed Systems*, Lisboa, Portugal, November 1991. Also AR/51-91.