On Stacks and Russian Dolls: Mobile Objects in Configurable Communication Protocols

José Orlando Pereira
jop@di.uminho.pt

Rui Oliveira
rco@di.uminho.pt

Departamento de Informática
Universidade do Minho, Portugal

Abstract. This paper introduces *Groupz*, a novel development framework for group communication protocol. *Groupz* merges advantages of traditional communication protocol support environments with object mobility, proposing multiple nested mobile objects as the natural evolution of layered protocols. By shifting the focus of protocol development from data messages to mobile objects, it makes possible to build configurable and adaptable system software, suited for problematic environments such as world-wide networks and mobile computers, without overlooking efficiency.

1 Introduction

Programming reliable distributed systems is certainly a complex task. Much of the difficulties are usually tackled by developing communication protocols that provide powerful abstractions, such as view synchronous process groups and totally ordered multicasts.

Usually, communication protocols and their applications are loosely coupled by an interface which provides generic message passing primitives and isolates the application programmer from the details and complexities of the underlying communication sub-system.

Nonetheless, two arguments call for the integration of application and protocol development to allow more control over communication resources. First, there are demanding applications, such as distributed shared memory and distributed object systems, that require fine grained control and customization of communication sub-systems to achieve good performance. Second, is the broadening of the computing base where support for reliable communication protocols is desired, such as mobile hosts and wide-area networks, which exhibit frequent partial failures, are highly heterogeneous, are dynamic and must scale gracefully to thousands of nodes.

This wide range of requirements calls for a distributed programming framework that is simultaneously appealing to the communication protocol developer and at least, customizable by the application developer, for whom key issues are the possibility to reconfigure the system both at compile time and run-time and to reuse existing components as often as possible.

In order to fulfill these requirements, this paper introduces *Groupz*, a novel distributed development framework that merges advantages of traditional communication protocol support environments with a flexible component architecture.

In *Groupz*, object mobility is used to shift the focus of the system developer from protocols to messages, trading data messages by nested mobile objects, which can enormously improve the opportunities for protocol customization by applications, while retaining the architectural advantages that made layered protocols popular.

This paper is organized as follows. In Section 2, four paradigmatic approaches to communication protocol development are briefly presented and discussed. In Section 3, a general component architecture is described and in Section 4 it is shown how it can be used in the development of communication protocols. In Section 5, a reliable communication service that takes advantage of the described architecture is presented. Section 6 concludes the paper.

2 Protocol development survey

2.1 Protocol stacks

The most widely used architecture for communication protocol development is the layered approach, as seen in the x-Kernel [12] and Horus [25]. These systems provide support for protocol development in the form of an object-based model for layer composition and a library of utility routines providing a set of shared abstractions.

Protocols are structured as stacks of modules which implement a single uniform interface. Each message traverses the protocol stack downwards if being sent or upwards if being received. Depending on the semantics of each layer, messages can be stored, delayed, or even dropped. Typically messages are modified, by adding or removing headers, and sent to the next layer.

This system structure has proven to be quite effective in modularizing protocol code in separate and interchangeable layers while allowing a performance level as good as what is achieved with monolithic implementations, which makes it particularly suited for system programmers.

Other advantage of this structure for large scale and heterogeneous networks is the possibility of building gateways at different levels of abstraction, either to hide complexity or to translate between functionally identical protocols.

On the other hand, the uniform interface is often complex and tied to a particular set of protocols. The different needs of different layers also mean that the interface is the aggregation of several distinct services (e.g. remote invocation and message passing in x-Kernel [6], or membership and message passing in Horus [26]) which seldom are all implemented in the same layer.

Attempts to use layers as fine-grained protocol components have also met some problems, resulting in violation of independence between layers and other problematic workarounds [16].

2.2 Event-driven micro-protocols

As protocol layers tend to be rather large components, event-driven micro-protocols have been proposed as a complementary approach to further subdivide communication protocols [11, 3].

In this model, composite protocol layers are structured as some shared data and a set of micro-protocols. Each of these is a collection of event handlers that operate on shared data and messages, register and deregister other event-handlers and fire-up events. Events can either be system defined, such as the arrival of a new message, or tailored for communication between specific micro-protocols.

This allows the implementation of independent abstract properties as separate software modules, that can be composed into meaningful protocols, which is a big advantage to application programmers wishing to build customized communication sub-systems [9, 10].

2.3 Protocol classes

An object-oriented alternative to protocol layering is the specialization and extension by inheritance of protocol classes, as used in BAST [5].

Protocols are implemented as different classes corresponding to different roles in a particular distributed protocol. These classes provide different interfaces for different roles such as point-to-point peers, multicast peers, remote invocation servers and clients or agreement initiators and participants.

The system provides a collection of generic classes that can be used either directly or as base classes for special purpose protocols required by each application. An example of this is the specialization of an abstract agreement protocol into either an atomic commitment or a totally ordered multicast protocol.

This system structure is particularly appealing to application programmers, as it allows communication related and application specific code to be tightly coupled.

Nonetheless, using inheritance to extend protocol classes also has some problems. For instance, it makes the separate reuse of protocol extensions difficult because extensions become tied to their base classes and can not be reused to extend other functionally similar base classes without re-compilation. Composition and delegation have been proposed as appropriate methods to address this and other problems [4].

2.4 Active networks

The concept of active networks emerges from the possibility of configuring computations done by network nodes on packets on a per user basis [22]. This capability is intimately related to the inclusion of mobile code in packets themselves, which can be installed and executed on foreign nodes.

An extreme approach is the usage of *capsules* [23] or *messengers* [15, 24] where every message is a program to be executed, moving most of protocol code

to messages themselves. This is roughly the equivalent of migrating a thread from the sender to the receiver for every message, carrying along related code and data. As such, they are particularly suited to special purpose protocols which make use of code mobility or when dynamic reconfiguration of the communication protocols is a must.

Although these are certainly the most flexible of all the architectures discussed, they impose some overhead on messages making it hard to implement different aspects of the same protocol, such as reliability and order in process groups, as separate software modules.

Low-level interfaces between messages and hosts may also compromise the possibility of evolving the network infrastructure by creating new kinds of nodes, such as an unanticipated gateway, without rewriting mobile code.

2.5 Discussion

All these different structuring methods reflect specific targets in modularity for reuse or configurability, and as such should be carefully evaluated before shaping a new system.

For instance, some of them target the integration of different services into a coherent whole, as is the case of layered protocols, while others aim to ease the task of building a single service, like micro-protocols.

It is also important to distinguish configurability of communication sessions from protocols that are configurable on a per message basis, which only messengers and capsules can do.

Finally, configuration of protocols can be dynamic, which is an important issue when upgrading large installed bases of users and applications and which is supported only in the context of active networks.

However, these approaches are not mutually exclusive and it should be possible to take advantage of the best features offered by each of them, trying to apply each where it fits best. This should overcome the problems faced with each of them separately.

3 Component framework

3.1 Overview

In order to take advantages of all protocol development practices described, *Groupz* is based on a simple object-oriented component architecture. This framework includes a set of guidelines, interfaces and utility classes that help the programmer to build compliant components and use them together in complex systems.

The *Groupz* component framework is designed to be both easy to use and efficient when performing those tasks which are expected to be needed in communication protocols. Its use is not however restricted to protocols and should be applicable to other problem domains.

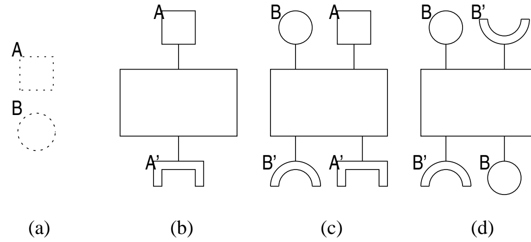

Fig. 1. Interfaces, features and components. (a) Two interfaces; (b) a component exporting two features, both a service and a dependency on interface A; (c) features with different interfaces on one component; (d) multiple features with the same interface on a single component.

3.2 Components

In *Groupz*, programs and data are partitioned and encapsulated in *components* and their functionality is abstracted as a set of related and cooperating *services*. In order to perform its function, a component may also have some *dependencies* on services provided externally. Exported services and external dependencies are together called the *features* of the component. Features are syntactically defined by *interfaces*, and thus the definition of a component is given by its set of interfaces.

A component may, as necessary, exhibit any number of features. This means that a component can have several features with the same interface and that features can be added and removed dynamically as appropriate.

Being an object-oriented framework, inheritance can be used to extend and specialize components, either by adding new features or by redefining existing services.

A dependency can be fulfilled by a matching service, creating a *link* between two components. Logically, two interfaces match if they are identical or if the service interface is an extension of the dependency interface. The link reflects a client-server relationship between two components and is itself described by the interface associated with the satisfied dependency. Other than these explicit inter-component relationships, which are properly documented as features, components must be fully self-contained.

Besides providing well defined and self-documenting software components, this standardization of relationships makes it possible to build higher level generic components, that include and manipulate sub-components.

3.3 Graphs

As a consequence of the variable number of features on components, each one can be connected to a variable number of other components. This means that a

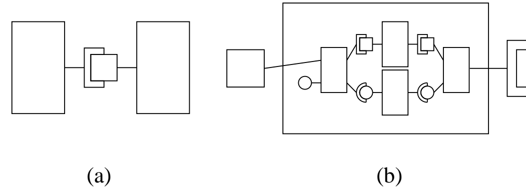

Fig. 2. Links and graphs. (a) Two components and a link; (b) a composite component exporting one service and one dependency.

complex system can be seen as a graph where components and links are nodes and edges, respectively.

Component graphs are themselves regarded as components. As such, recurring sub-graphs in complex systems can be encapsulated and transparently reused. Essential to the success of this strategy, is the possibility to export selected features of sub-components as features of the complex component itself.

Depending on the specific composition strategy used, the internal structure of these components can be defined in several ways, eg.:

- simple static collections of sub-components, which support *mix-in* style composition by delegating different features on different sub-components;
- static graphs, structured as a predefined set of components and links, useful to hide complexity and separate different levels of abstraction;
- dynamic graphs, that create and destroy components and links on demand from a template description;
- incomplete graphs, which are completed with components dynamically provided as necessary.

All these composite components can also be generic or specific, in the sense that their internal structure is fully programmable or reflects roles in a design or architectural pattern. Components that allow their internal structured to change dynamically often advertise this possibility as an exported service of the composer itself. This allows a controller component to be linked to this service in order to manipulate the graph.

3.4 Applying the framework

In order to apply the general component architecture described above to a specific problem domain, three tasks have to be performed:

i. define a set of interfaces that capture the syntax of client-server relationships between the entities in the considered problem domain;
ii. build a set of, possibly abstract, components corresponding to the entities identified, using the appropriate interfaces to shape the exported features;

Data-flow	Specifies a data-flow target. It is used both for data-flow between protocol layers as well as for event delivery.
Control-flow	Specifies that the object is runnable. Examples of usage are timers, device drivers and mobile components. In Java this is just the `java.lang.Runnable` interface.
Dictionary	The original `java.util.Dictionary` class provides access to lookup tables.

Table 1. Common service interfaces for protocol development.

iii. identify recurring graphs or different abstraction levels and implement them as composite components.

The resulting domain specific framework can then be used by an application developer, who will configure graphs as needed, possibly using new or extended application specific components.

4 Protocol framework

4.1 Overview

The basic assumption of the *Groupz* protocol framework is that all entities, including protocols and messages, are components as defined by the component framework.

This fact is the single most important feature of the proposed architecture, as it lays the foundation for shifting complexity from statically configured protocols to message carriers that can be dynamically selected and parameterized by the client application. The abstraction of messages as components is made possible by the implementation in the Java programming language, which provides seamless object [19, 21] and code mobility [14].

Multiple nested message carriers are proposed as the preferred architecture for developing configurable complex communication protocols, unifying most of the advantages of traditional protocol development environments with new features introduced by object mobility.

4.2 Protocols as components

Extending the concept of protocol stacks to component graphs, connected by the small set of simple interfaces presented in Table 1, obviates most of the difficulties found when reusing layered protocols, such as hidden dependencies, which are largely related to their complex uniform interfaces and large granularity.

Multiple simple interfaces result in small self-contained software modules. As a consequence, protocol abstraction layers can be themselves fragmented into graphs of simple components, instead of being monolithic layers. As these small components tend to solve recurring abstract problems, they are reusable in more situations than more complex protocol layers, regardless of the uniform interface of the later, because there are no hidden dependencies between them.

On the other hand, the reuse of complex layers themselves is eased by the fact that multiple features per component specify as many services and explicit dependencies as necessary. As such, what would be a hidden dependency between two layers to comply with an uniform interface becomes an explicit and separately manageable feature of the component.

4.3 Messages as components

The implications of also abstracting messages as components are certainly more profound. Network data formats and buffer management, which traditionally are big concerns in protocol development, become irrelevant. Adding headers is abstracted as object composition and serialization is done all at once by a dedicated node in the protocol graph, that accepts serializable components and produces byte sequences.

Traditional protocols operate on non-encapsulated data, so headers inserted by a protocol layer must be read only by the same layer. Since, in *Groupz* both messages and protocols are components related only by complementary sets of exported features, protocols do not need to be aware of the internal structure of messages and independent implementations of both can be developed.

As a consequence, it is possible to move most of the complexity from protocol implementations to the messages themselves. A protocol layer becomes just the provider of some features that messages use. Different implementations for messages can then be assembled independently of the protocol layer, as long as the agreed interfaces are respected.

In short, a specific protocol is implemented by a generic *protocol host component* and a set of *message carrier components*, related by their complementary features. An application willing to send a message (see Figure 3), wraps it with an adequate carrier and sends it directly to the lower network layer. When it arrives at its destination, the carrier is linked to the host component so the delivery can be negotiated.

The negotiation between the host and the carrier may involve a series of transactions, depending on the system being implemented. The success of this strategy depends on the features of both carriers and hosts, as they have to be generic enough to support a range of implementations, while being efficient at least when performing the most common protocols. As such, it is important that they describe abstract services provided and requested by carriers and hosts and not implementation details which will restrict the possibilities of evolution.

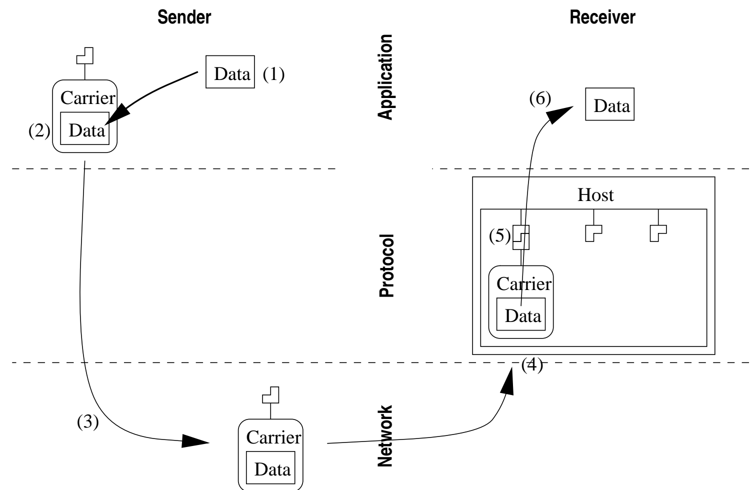

Fig. 3. An application willing to send a message (1), wraps it with an adequate carrier (2) and sends it directly to the network (3). When it arrives at its destination (4), the carrier is linked to the host component so the delivery can be negotiated (5) and eventually happens (6).

4.4 Nested carriers

The need to define interfaces between carriers and hosts that are both generic and efficient can be addressed by separately enforcing different aspects of a protocol. Consequently, host components do not need to support every conceivable protocol and most optimization strategies that are known from monolithic protocol implementations can be reused with success.

This decomposition is notably similar to the process of partitioning traditional protocols as layers and also results in a stack of components. Consequently, an application willing to send a message, has to use multiple nested carriers to wrap it (see Figure 4). When arriving at the destination, the outer layer will be connected to the lower host component and will eventually release its load after the required negotiation. The delivered component is the carrier that will proceed to next protocol host which is exactly one layer up. An analogy can be made to a russian *matrioshka* doll being opened, layer by layer, until the last one is reached.

In addition, as happens with traditional layered architectures, developers can take advantage of partitioning to build large hierarchical networks by using gateways to connect individual sub-networks at different levels. This possibility is essential for efficient protocols in the context of wide-area networks.

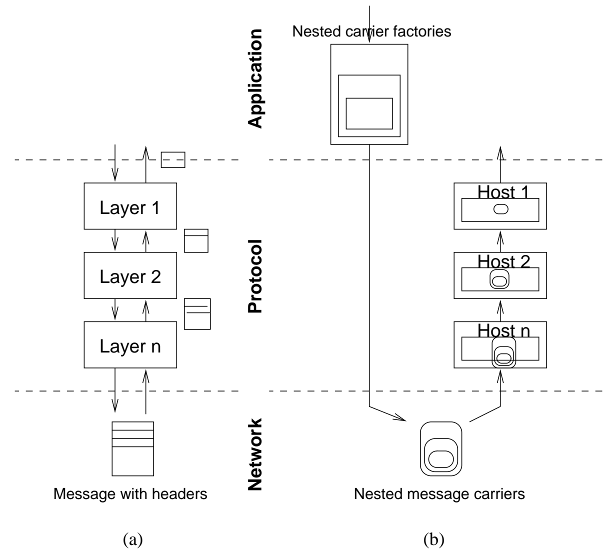

Fig. 4. Layered protocols and nested carriers. (a) Layered protocol stack generating a message with protocol specific headers; (b) protocol hosts export public interfaces for "russian doll" carriers, built at application level.

4.5 Discussion

It is interesting to examine how this architecture uses proven concepts from existing protocol development environments and architectures, besides the obvious similarities with layered protocols.

Event-driven micro-protocols and protocol classes, resemble two different techniques that have been used with success to build and extend carrier and host components. In fact, it is a straightforward process to convert micro-protocols to carriers, by aggregating event-handlers by message and not by protocol.

A carrier can also be considered an hybrid solution between messengers or capsules and fixed messages, as protocols can also be customized on a per message basis but no code migration is done by default. However, if dynamic reconfiguration is necessary the object-oriented framework proposed is easily extended by class carriers, which make use of the underlying Java infrastructure in code mobility and security to remotely install carrier implementations.

Groupz emphasizes an object-oriented design and implementation of complex communication protocols, by concentrating on the development of carrier to host interfaces regardless of the kind of protocol being implemented. A key issue for this is the possibility of nesting carriers and layering hosts to allow the decomposition of complex interactions in generic services.

In fact, given appropriate serialization layers, *Groupz* can be used for a wide range of protocols. For instance, an active networking system can be devel-

oped by using a serialization layer that appends appropriate code for classes being sent. Even protocols compatible with traditional implementations, such as TCP/IP, can be developed by using a serialization layer that maps objects to standardized packet structures.

A different way to look at *Groupz* is to point out that it opens the implementation [13] of protocols to application programmers by separating policy and mechanism and allowing the definition of the former by carrier components.

5 Case Study

5.1 Overview

The *Groupz* project aims at building a set of communication protocols for reliable distributed application development for large-scale networks based on process groups [2]. Communication services based on this abstraction usually offer reliable multicast services, message ordering services and group management services [8] providing dependable message delivery and consistent failure reporting.

To show how the protocol framework is used, two aspects of a reliable communication system are examined. It is shown how to take advantage of a generalized protocol graph to configure a virtual unreliable multicast network and it is shown how a configurable dependable delivery protocol can be built using message carrier and host components.

5.2 Environment

Wide-area networks pose several challenges when compared to local-area networks due to both geographical separation and number of sites [1]. Geographical large scale networks are unreliable in the sense that they introduce unpredictable delays and may drop or duplicate messages. Link failures may also occur, leaving the network partitioned for noticeable periods of time. Numerical large scale is another challenge, as applications may require groups including a large number of members. These networks also tend to be highly heterogeneous, encompassing nodes of various manufacturers and computational power, ranging from handheld portables to large servers.

To address these challenges, the developer must be able to state the minimum requirements of the application as accurately as possible, in order not to incur in unnecessary overhead. For instance, requiring reliable message delivery in the presence of frequent and long lasting network partitions, results in having to store messages for retransmission for possibly long periods of time. In order to minimize the amount of storage required, it should be possible to discard messages that become obsolete while waiting for retransmission.

Some proposals in this area exist [7, 20]. However, they tend to be customized to particular applications. The architecture introduced by *Groupz* allows applications to integrate these and other solutions and select which is appropriate for each individual message.

5.3 Unreliable multicast

The fundamental service to build a group communication protocol is the abstraction of an unreliable multicast network, spreading messages to whoever is listening on the appropriate channel. This is an example of a service that does not need to be customized for each message, and as such, does not make use of carriers. It is nonetheless configurable by choosing an appropriate structure for the graph from a set of existing components.

For instance, if a true multicast network is not available, it is simulated on top of point-to-point networks by using an approximate membership for the group. It is even possible to configure the system as a combination of both, as any multicast service, real or simulated, can be used as a single connection under the simulated multicast component.

Being the lower layer of the system, in *Groupz* it must also perform object serialization and packaging as network data units. Depending on the network, this may require fragmenting and reassembling.

In addition, site failure suspicion as is required by some distributed algorithms is done at this layer, by inserting some extra messages in the network. This is done in one of two different ways, either by monitoring regular heartbeats from every site or by challenging sites that are suspected to be down or unreachable.

5.4 Dependable delivery

The dependable delivery service is expected to perform buffering and retransmission as appropriate to ensure ideally exactly-once atomic delivery. As the cost of doing this in large-scale networks is prohibitive, the requirements have to be relaxed, and as such, control has to be given to the application through the use of custom message carriers.

In order to know if a message is to be delivered to a host component, a carrier requires information about the location where it is and about what messages have already been delivered both locally and remotely. This information is also used to decide if they need to retransmit or discard themselves.

Although location information is static, information about message delivery is dynamic and has to be updated at different locations. As this involves communication, it is also done by carrier components, that are generated by message carriers when appropriate. This is analogous to the use of acknowledge messages in traditional protocols.

With these tools it is quite easy to supply different qualities of service just by modifying carrier components, specifically, changing the conditions upon which they retransmit or discard themselves and deliver their load based on available information. Currently, reliable delivery to a group, either safe or not, is implemented along with stubborn and selective overlapping messages, which by faking acknowledges as necessary, allow respectively all or some their predecessors to discard themselves.

An interesting message carrier is the one associated with group membership changes under a virtually synchronous environment. This event often means that some messages are discarded from retransmission buffers, even if not fully delivered. To accomplish this, traditional group protocols usually have specific control operations, which either circumvent the uniform protocol interface or are contained in it, making it more complex. In *Groupz* this is not necessary, as the group membership change message, itself, acts as an universal acknowledge from failed sites when reaches the dependable delivery host, discarded messages that are no longer needed.

6 Conclusions

In the paper, we argue that existing protocol development tools, in isolation, are unsatisfactory for the development of complex highly configurable protocols. As a result, *Groupz* combines most of the advantages of traditional protocol development environments with new features introduced by object mobility into a coherent and flexible protocol framework. This framework helps the programmer to describe the relationships between protocol components as services and dependencies, making them separately reusable while encouraging interface and class inheritance where appropriate for extension and customization.

We show that it is a valid assumption to consider both protocols and messages as opaque components. The abstraction of messages as components is in fact the single most important feature of the proposed architecture, as it lays the foundation for shifting most of the complexity from statically configured protocols to the messages themselves. Multiple nested message carriers are proposed as an adequate extension of the architecture for structuring configurable complex communication protocols. This strategy allows applications to dynamically select and parameterize complex communication services.

The *Groupz* component and protocol frameworks [17, 18] have been implemented in the Java programming language and are currently being used to support a set of reliable communication protocols based on the process group abstraction over large-scale networks. In addition to the virtual network and dependable delivery layers, *Groupz* includes ordering and membership layers and an agreement service, offering a complete range of configurable group communication services.

References

1. Ö. Babaoglu and A. Schiper. On group communication in large-scale distributed systems. In *Proceedings of the 6th SIGOPS European Workshop*, September 1994.
2. Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 1993.
3. Henrique Jorge da Fonseca. Ambientes de suporte para modularização, concretização e execução de protocolos de comunicação. Master's thesis, Universidade Técnica de Lisboa, Instituto Superior Técnico, 1994.

4. R. Guerraoui et al. Strategic research directions in object oriented programming. *ACM Computing Surveys*, 28(4):691.
5. B. Garbinato, P. Felber, and R. Guerraoui. Protocol classes for designing reliable distributed environments. In *Proceedings of ECOOP'96*, July 1996.
6. Network Systems Research Group. *x-Kernel Programmers Manual*. Dept. of Computer Science, University of Arizona, January 1996. Version 3.3.
7. R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical report, LSE, EPF Lausanne, December 1996.
8. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Computer Science Department, Cornell University, 1994.
9. M. Hiltunen and R. Schilchting. Understanding membership. Technical Report TR95-07, Dept. of Computer Science, University of Arizona, 1995.
10. M. Hiltunen and R. Schlichting. A configurable membership service. Technical report, Department of Computer Science, University of Arizona, 1994.
11. Matti Aarno Hiltunen. *Configurable Fault-Tolerant Distributed Services*. PhD thesis, Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, July 1996.
12. N. Hutchinson and L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
13. G. Kiczales and Xerox Parc. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, January 1996.
14. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.
15. G. Di Marzo, M. Muhugusa, C. Tschudin, and J. Harms. The messenger paradigm and its implications on distributed systems. In *Proceedings of ICC'95 Workshop on Intelligent Computer Communication*, 1995.
16. S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1(2):87–103, December 1993.
17. José Orlando Pereira. Groupz component framework. Technical report, University of Minho, Departamento de Informática, Campus de Gualtar, 4710 Braga, Portugal, June 1997. http://gsd.di.uminho.pt/~jop/#tr.
18. José Orlando Pereira. Groupz protocol framework. Technical report, University of Minho, Departamento de Informática, Campus de Gualtar, 4710 Braga, Portugal, June 1997. http://gsd.di.uminho.pt/~jop/#tr.
19. R. Riggs, J. Waldo, A. Wollrath, and K. Bharath. Pickling state in the Java system. *Usenix Computing Systems*, 9(4):291–312, Fall 1996.
20. L. Rodrigues and P. Veríssimo. How to avoid the cost of causal communication in large-scale systems. In *Proceedings of the 6th SIGOPS European Workshop*, September 1994.
21. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043. *Java Object Serialization Specification*, December 1996. 1.2.
22. D. Tannenhouse, J. Smith, W Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications*, 35(1):80.
23. D. Tannenhouse and D. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), April 1996.

24. C. Tschudin, G. Di Marzo, M. Muhugusa, and J. Harms. Messenger-based operating systems. *Cahier du Centre Universitaire d'Informatique*, July 1994. Revised September, 1994.
25. R. van Renesse, K. Birman, B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Cornell University, Computer Science Department, March 23 1995.
26. Robbert van Renesse. The Horus uniform group interface. Technical report, Cornell University, 1996.

This article was processed using the LaTeX macro package with LLNCS style