A.5 Syntax of Emerald—Changes

operationDefinition ::= [**private**] *operationSignature*
[**by** *identifier* " : " *typeDefinition*]
blockBody
end *operationName*

statement ::= ... |
broadcastStatement |
... |
quitStatement |
...

broadcastStatement ::= **broadcast** *invocationMessage* **on** *expression*
[**to** *identifier* " : " *typeDefinition*]
initialProcess
[*replySection*]
[*unavailableHandler*]
[*failureHandler*]
end broadcast

invocationMessage ::= *operationName argumentList*

replySection ::= **reply** [*declarationList*] **received**
[**last** *identifier* " : " *typeDefinition*]
blockBody
end reply

declarationList ::= *singleDeclaration* |
"[" *listOfDeclarations* "]"

listOfDeclarations ::= *singleDeclaration*
singleDeclaration " , " *listOfDeclarations*

singleDeclaration ::= *identifier* " : " *typeDefinition*

quitStatement ::= **quit** [**when** *expression*]

initialProcess ::= *blockBody*

```
const Channel ==
   immutable object Channel
      export Of
      function Of[MemberType : AbstractType] → [Result : ChannelCreatorType]
         where
            ChannelCreatorType ==
               type ChannelCreatorType
                  function GetSignature → [Signature]
                  operation Create → [OpenChannelType]
               end ChannelCreatorType
            OpenChannelType ==
               type OpenChannelType
                  operation Subscribe[MemberType]
                  operation UnSubscribe[MemberType]
                  function GetMembers → [Vector.Of[MemberType]]
               end OpenChannelType
         end where
         Result ←
            immutable object ChannelCreator
               export GetSignature, Create
               function GetSignature → [Result : Signature]
                  Result ← OpenChannelType
               end GetSignature
               operation Create → [Result : OpenChannelType]
                  Result ←
                     object OpenChannel
                        export Subscribe, UnSubscribe, GetMembers
                        operation Subscribe[Member : MemberType]
                           ...
                        end Subscribe
                        operation UnSubscribe[Member : MemberType]
                           ...
                        end UnSubscribe
                        function GetMembers → [Result : Vector.Of[MemberType]]
                           ...
                        end GetMembers
                     end OpenChannel
               end Create
            end ChannelCreator
      end Of
   end Channel
```

A.4 Built-in Objects

A.4.1 Channel

The *Channel* built-in object implements basic group maintenance operations. The *Of* operation of the *Channel* object takes a parameter of the type *AbstractType* and returns a channel creator object. The *Create* operation of a channel creator returns an open channel object with an associated type given by the corresponding *Of* invocation. An open channel identifies a group i.e. a set of objects which participate together in the group communication initiated on the channel. A group communication is accomplished by concurrent invocation of all members of a group by the same message broadcast via a channel.

An open channel may be used in a **broadcast** statement in the way described in the section **Broadcast Statement**. When a method is used with a channel in a **broadcast** statement it is invoked concurrently on all objects which subscribed to the channel.

The *Subscribe* and *UnSubscribe* methods of an open channel object maintain a member list of a group identified by a channel. The *Subscribe* method allows objects to become members of a group identified by an open channel and subsequently receive all messages broadcast via this channel. The *UnSubscribe* method permits to withdraw membership of an object from a group. The *GetMembers* function gives access to the list of current members of a group identified by an open channel.

channel used in the section and arguments of the invocation, as well as failure of locating member objects may be detected by this handler. A **reply** section and initial process are nested in the block created by a **broadcast** statement so an uncaught failure in any of them may be detected by a handler in the **broadcast** statement.

Any exception raised by execution of a **broadcast** statement and not caught inside the statement causes the whole statement to fail and is equivalent to executing a sequence of statements **quit** and **assert false**. All subsequent replies are discarded; execution of the **reply** section for already received replies and the block of the statement are finished normally except for the thread of control which failed. The failure is propagated to the embracing block.

A.2 The Quit Statement

The **quit** statement terminates the textually innermost **broadcast** statement and is invalid outside such a statement.

quitStatement ::= **quit** [**when** *expression*]

The expression following the **when** keyword is of type *Boolean*.

The simple **quit** statement terminates the **broadcast** statement unconditionally. The conditional **quit** statement (i.e. with the optional **when** clause) terminates it if the *expression* evaluates to **true**. If it evaluates to **false**, the **quit** statement is equivalent to an empty statement.

A.3 Operation Definition

By clause

An operation may be invoked either directly on the object implementing it or indirectly via an open channel object. In the latter case, a number of objects may be invoked concurrently accomplishing in this way group communication. To obtain information about a group of objects involved in the same broadcast invocation an optional **by** clause is added to the operation definition.

operationDefinition ::= [**private**] *operationSignature*
[**by** *identifier* " : " *typeDefinition*]
blockBody
end *operationName*

The **by** clause defines a variable of type *Vector.Of*[*Any*]. If the object was invoked directly then the variable contains a reference to the *NIL* object. If the object was invoked by use of **broadcast** statement then the variable contains a list of objects which received the same broadcast message.

Concurrency Control in the Broadcast Statement

As a result of initiating the **broadcast** statement all objects which subscribe to the appropriate channel are invoked. In each of them a new process is started asynchronously to perform the invoked method. When the process finishes its operation in the invoked object then the thread of control is returned to the **reply** section in the broadcasting object. All the new processes return asynchronously.

To prevent the asynchronous processes inside the **reply** section and in the initial process from interfering with each other access to variables visible in the scope of the **broadcast** statement has to be restricted. If a **broadcast** statement is placed in a monitored part of an object then the processes inside **broadcast** statement are serialized and have exclusive access to all variables. If the statement is placed outside the monitored part then a process inside the statement have full access only to the variables declared locally in the part of the statement it is executing (a **reply** section or an initial process) and a read access to variables defined outside of it.

Terminating the Broadcast Statement

The **broadcast** statement may be terminated in one of three ways: execution of the **quit** statement, execution of a statement terminating execution of a block embodying the **broadcast** statement or by a failure.

The normal way of terminating the **broadcast** statement is by execution of the **quit** statement inside it. The process executing the **quit** statement is terminated. All subsequent replies to the broadcast invocation are discarded. Execution of the initial process of the statement and of the **reply** section for already received replies is finished normally except for the process which executed the **quit** statement. The suspended process that initiated execution of the **broadcast** statement is restarted and it continues execution of the operation the **broadcast** statement is embedded in. Execution of the statement following the **broadcast** statement is started immediately without waiting for the other processes to finish.

The statements: **return**, **returnandfail** or **exit** corresponding to a loop containing the **broadcast** statement may be executed inside the **broadcast** statement. They are equivalent to executing the **quit** statement and then appropriate statement outside the **broadcast** statement.

The **broadcast** statement may fail and thus terminate its execution in the way described below.

Failure Handling

A failure in an initial process of a **broadcast** statement may be caught by an appropriate handler inside the block of the process. A failure caused by execution of a **reply** section may be caught by a handler inside the section. A failure in executing a broadcast invocation is propagated backward from the callee to the object executing the **broadcast** statement and may be caught by a handler placed after the **reply** section in the **broadcast** statement. Also exceptions caused by evaluation of expressions describing an open

- expressions determining arguments are evaluated,
- invocation message consisting of the method name and arguments is passed to all objects being members of a group identified by the open channel,
- the process that executes the broadcast statement is suspended until a **quit** statement is executed in the **broadcast** statement,
- the broadcast invocation message is delivered to all member objects in the atomic causal order, i.e., all objects in a system receive broadcast invocation messages in the same order, and if an invocation is caused by another invocation then the second one is always started in all invoked objects before the first one.
- a new process is created for each member object,
- the operation is invoked on the member objects by the new process,
- a list of the member objects is assigned to a constant defined by the **to** clause,

Initial Process

After the broadcast invocation has been started in all the member objects, a new process is created which executed the initial process of the **broadcast** statement.

Reply Section

Results of a broadcast invocation may be collected and processed in the **reply** section of the corresponding **broadcast** statement.

replySection ::= **reply** *declarationList* **received**
[**last** *identifier* " : " *typeDefinition*]
blockBody
end reply

The list of declarations bracketed by keywords **reply** and **received** defines variables which contain results of a broadcast invocation. The number of variables in the list must be equal to the number of results of the method from the *invocationMessage* and the types of the results must positionally conform to the types from the list.

The optional clause **last** allows to identify the last result of a broadcast invocation. The declaration following the keyword **last** defines a variable of type Boolean and is initialized to **true** if the returned result is the last expected one or to **false** otherwise.

The **reply** section is executed separately for all replies. When a reply is received a list of variables is initialized with corresponding results, the variable defined by the **last** clause is initialized (if required), and the block of the section is executed.

Appendix A

Changes to the Emerald Language Report

A.1 The Broadcast Statement

Group communication is accomplished by execution of the **broadcast** statement. Upon execution of the statement, a group of objects is invoked concurrently, the appropriate method on each of them is executed, and the results of the invocations may be collected by the invoker.

Broadcast Invocation

An invocation message may be broadcast to a number of objects by use of an open channel object in a **broadcast** statement.

broadcastStatement ::= **broadcast** *invocationMessage* **on** *expression*
[**to** *identifier* " : " *typeDefinition*]
initialProcess
[*replySection*]
[*unavailableHandler*]
[*failureHandler*]
end broadcast

The expression following the **on** keyword is an open channel, i.e., an object of type *Channel.Of*[*MemberType*] where *MemberType* is an abstract type associated with the open channel. An open channel is an object which identifies a group of object. The *invocationMessage* is a name of an operation defined in the type *MemberType* and a list of actual parameters of the method. The operation is invoked with the given arguments in all objects that belong to a group identified by the open channel. The optional clause **to** contains a declaration of a variable of type *Vector.Of*[*MemberType*] whose scope is the whole **broadcast** statement. The variable contains a list of all invoked objects.

A broadcast invocation is accomplished in the following steps:

- the expression which determines an open channel is evaluated,

[Raj 91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A General-Purpose Programming Language. *Software — Practice and Experience* 21(1):91–118, January 1991.

[Ricciardi 91] Aleta M. Ricciardi and Kenneth P. Birman. *Using Process Groups to Implement Failure Detection in Asynchronous Environments.* Technical Report TR-91-1188, Cornell University, February 1991.

[Ritchie 74] D.M. Ritchie and K. Thomson. The UNIX Time-Sharing System. *Communications of the ACM* 17(7):365–375, July 1974.

[Saltzer 84] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems* 2(4):277–288, November 1984.

[Satanarayanan 90] Mahadev Satanarayanan and Ellen H. Siegel. Parallel Communication in a Large Distributed Environment. *IEEE Transactions on Computers* 39(3):328–348, March 1990.

[Shoch 82] John F. Shoch and Jon A. Hupp. The "Worm" Programs—Early Experience with a Distributed Computation. *Communications of the ACM* 25(3):172–180, March 1982.

[Spector 86] A.Z. Spector, D. Daniels, D. Duchmap, J.L. Eppinger, and R. Paush. Distributed Transactions for Reliable Systems. In *Proc. of the Tenth ACM Symposium on Operating Systems Principles*, pages 127–146. Orcas Island, Washington, 1986.

[Tanenbaum 89] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, second edition, 1989.

[Lamport 78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21(7):558–565, July 1978.

[Leffler 89] Samuel J. Leffler and Marshall Kirk McKusick. *The Design and Implementation of the 4.3 BSD Unix Operating System.* Addison Wesley, 1989.

[Liang 90] Luping Liang, Samuel T. Chanson, and Gerald W. Neufeld. Process Groups and Group Communication: Classification and Requirements. *Computer* 23(2):56–66, February 1990.

[Luan 90] Shyh-Wei Luan and Virgil D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Transactions on Parallel and Distributed Systems* 1(3):271–285, July 1990.

[Makpangou 91] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Mark Shapiro. Fragmented Objects for Distributed Abstractions. October 1991. to appear in IEEE Software.

[MelliarSmith 90] P.M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems* 1(1):17–25, January 1990.

[Metcalfe 80] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM* 19(7):395–404, July 1980.

[Notkin 90] David Notkin. Proxies: A Software Structure for Accommodating Heterogeneity. *Software — Practice and Experience* 20(4):357–364, April 1990.

[Pardyak 92a] Przemysław Pardyak. Group Communication in an Object-Oriented Environment. In *Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems I-WOOOS'92.* IEEE, Paris, France, September 1992.

[Pardyak 92b] Przemysław Pardyak. Group Communication in Emerald. An Example of Application. February 1992. name server using group communication.

[Peterson 89] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems* 7(3):217–246, August 1989.

[Gray 78] Jim Gray. Notes on Data Base Operating Systems. In G.Goos and J.Hartmanis (editors), *Lecture Notes in Computer Science*, Chapter 3.F., pages 394–481. Volume 60. Springer Verlag, 1978.

[Halpern 90] Joseph Y. Halpern and Yoram Moses. Knowledge and Common Knowledge in a Distributed Environment. *Journal of the ACM* 37(3):549–587, July 1990.

[Hoare 78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM* 21(8):666–677, August 1978.

[Hoare 87] C.A.R. Hoare. Monitors: an operating System Structuring Concept. *Communications of the ACM* 17(10):549–557, October 1987.

[Horowitz 86] Elis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures in Pascal*. Computer Science Press, second edition, 1986.

[Hutchinson 87a] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, January 1987. Technical Report 87-01-01.

[Hutchinson 87b] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. *The Emerald Programming Language Report*. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, Washington, October 1987. Also available as DIKU Report no. 87/22, Dept. of Computer Science, University of Copenhagen, Copenhagen, Denmark and as TR no. 87-29, Dept. of Computer Science, University of Arizona, Tucson, Arizona. (Revised August 1988).

[Jarvinen 90] H.-M. Jarvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systa. Object-Oriented Specification of Reactive Systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. 1990.

[Jul 88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems* 6(1):109–133, February 1988.

[Kaashoek 89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flyn Hummal, and Henri E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review* 23(4):5–19, October 1989.

[Kaashoek 91] M. Frans Kaashoek and Andrew S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *Proceedings of the 11th International Conference on Distributed Computer Systems*, pages 222–230. Arlington, VA, May 1991.

[Chandy 85] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of a Distributed System. *ACM Transactions on Computer Systems* 3(1):63–75, February 1985.

[Chang 84] Jo-Mei Chang and N.F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems* 2(3):251–273, August 1984.

[Cheriton 85] David R. Cheriton and Willy Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems* 3(2):77–107, May 1985.

[Clark 85] David D. Clark. The Structuring of Systems Using Upcalls. In *Proc. of the 10th ACM Symp. Operating Systems*, page . December 1985.

[Cooper 86] Eric C. Cooper. Replicated Procedure Call. *Operating Systems Review* 20(1):44–56, January 1986.

[Cristian 85] F. Cristian., H. Aghili, H.R. Strong, and D. Dolev. Atomic Broadcast: From Simple Diffusion to Byzantine Agreement. In *Proceedings of the 15th International Annual Symposium on Fault-Tolerant Computing Systems*, pages 200–206. IEEE, Washington, D.C., 1985.

[Dolev 87] Danny Dolev, Cynthia Dvork, and Larry Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM* 34(1):77–97, January 1987.

[Fisher 85] Michael J. Fisher, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM* 32(2):374–382, April 1985.

[Fisher 86] Michael J. Fisher, Nancy A. Lynch, and Michael S. Paterson. Easy Impossibility Proofs for Distributed Consensus Problems. *Distributed Computing* 1(1):26–39, January 1986.

[GarciaMolina 91] Hector Garcia-Molina and Annemarie Spauster. Ordered and Reliable Multicast Communication. *ACM Transactions on Computer Systems* 9(3):242–272, August 1991.

[Gehani 84] Narain M. Gehani. Broadcasting Sequential Processes (BSP). *IEEE Tr. on Software Engineering* SE-10(4):343–351, July 1984.

[Gligor 80] V.D. Gligor and S.H. Shattuck. Deadlock Detection in Distributed Systems. *IEEE Transactions on Software Engineering* SE-6(5):435–440, September 1980.

[Goscinski 91] Andrzej Goscinski. *Distributed Operating Systems, The Logical Design.* Addison-Wesley, 1991. ISBN 0-201-41704-9.

Bibliography

[Aspnas 89] M. Aspnas, R.J.R. Back, and R. Kurki-Suonio. *Efficient Implementation of Multi-process Handshaking on Broadcast Networks.* Reports on Computer Science A75, Åbo Akademi University, Turku, Finland, March 1989.

[Birman 87a] Kenneth P. Birman and Thomas A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. *Operating Systems Review* 21(5):123–138, November 1987.

[Birman 87b] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems* 5(1):47–76, February 1987.

[Birman 91a] Kenneth P. Birman and R. Cooper. The ISIS project: Real Experience with a Fault Tolerant Programming System. *Operating Systems Review* , April 1991.

[Birman 91b] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. *Design Alternatives for Process Group Membership and Multicast.* Technical Report TR-91-1257, Dept. of Computer Science, Cornell University, December 1991.

[Birman 91c] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems* 9(3):272–314, August 1991.

[Black 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86. October 1986.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering* 13(1):65–76, January 1987.

[Black 91] Andrew Black and Norman Hutchinson. *Type-checking polymorphism in Emerald.* Technical Report, Department of Computer Science, University of British Columbia, 1991.

experiments with group-based programming

Only a practical experiment may verify new ideas and methods. Therefore, shifting to the more abstract level should be preceded by more thorough investigation into programming with groups. More practical experience is necessary to be able to isolate the most important issues. A number of different size projects using group communication should reveal potential weaknesses of our approach and enable their removing before implementing enhanced mechanisms.

formal verification of protocols' correctness

Increasing complexity of distributed algorithms and communication protocols makes their construction and verification hard. Formal methods developed for specification and correctness proving may be effectively used for such protocols. It would be interesting to describe our protocols in one of the formalisms (e.g., LOTOS) and formally verify their properties.

experiment with different broadcast protocols

Experimental comparison of different broadcast protocols is an interesting issue which have not been fully investigated yet. Different protocols are developed for different systems and it is difficult to normalize results of measuring them. Our new IPC module implements different methods of broadcast in the same environment so it may be effectively used for experiments aimed at comparison of different methods and verifying impact of their application of a system performance.

Wide Area Emerald

Some preparations are being done to investigate possibilities of porting Emerald to Wide Area Networks. During the porting, all restrictions imposed on Emerald (e.g., one kernel per site) should be removed, and support for WAN communication and multicast should be added. Results of experiments described above should be applied in development of the new module.

security in Emerald

In the first implementation of Emerald, no concern was given to problems of security: no protection is given to objects, network communication is insecure. Extension of Emerald for Wide Area Networks forces implementors to take care about security because Emerald users become more exposed to security threats.

name server in Emerald

The Emerald operating system lacks a name service. In the first phase of this project, we have considered building a restricted name server based on registering channels in Emerald nodes. The idea of the channel registry turn out to be too restrictive and was dropped. The considerations revealed, however, necessity of such a service and showed some methods of its implementation. Having implemented broadcast protocols, we are able to use them for an efficient and simple name server.

ing group communication among objects. MOI is based on a simpler version of the general model of group structure. The simplified model and MOI are suited for object-oriented world. MOI is defined in terms of objects: objects are its components, and object invocations are used for communication. We have applied the Multiple Object Invocation paradigm for creating new group communication primitives for the Emerald object-based programming language. Channel objects and the broadcast statement comprise a flexible toolkit for programs using group communication. The new language constructs were verified practically by writing a number of short programs using them.

11.2 Contribution

Our contribution in the field of group communication was presented in this thesis and includes:

- a general model of a group structure and interactions,
- Multiple Object Invocation—a method for accomplishing group communication among objects,
- a new suite of broadcast protocols,
- the System State Agreement procedure—a protocol verifying membership in a distributed system,
- Virtual Reliability—an approach to reliability in distributed systems, and
- implementation of group communication in the Emerald object-based programming language and distributed system.

11.3 Further Research

Group communication is a wide field of computer science which opens possibilities for many research projects. Some of the possible directions of research taking our project as a starting point include:

specification of groups

Multiple Object Invocation and its Emerald implementation—channels and the broadcast statement—express group communication: interactions involving a group of object. The next step, leading to a higher level of abstraction, should allow specification of a group. Such specification could be used for creating a group on the basis of definition of membership and group communication protocols, methods for failure handling, and schemes of recovery from crashes. A group should be a single object encapsulating the protocols and methods, and interfacing clients to member objects.

Chapter 11

Conclusion

We have investigated issues related to group communication and applied results of the research in adding broadcast communication to the Emerald object-based programming language and distributed operating system. Results of our research have been described in this thesis.

11.1 Summary of the Thesis

The thesis describes our research in the field of group communication. We address issues at three levels: network protocols, system structure, and programming languages.

We were concerned about two problems at the level of a network: reliable broadcast and maintaining consistency in a distributed system. The goal was to create an efficient and reliable interprocess communication module that could be used in the Emerald system. We have achieved the goal: a new IPC module was implemented and it is used for language-level broadcast communication. The broadcast protocols we have devised are efficient and flexible. They provide a spectrum of services for varying demands for ordering and reliability. The System State Agreement procedure, integrated with the broadcast protocols, maintains consistency among nodes of a system and permits reliable communication in spite of node crashes. In the new IPC module we exploit the idea of Virtual Reliability—it adds to efficiency by optimistic approach to failures, and, in case of applications with lesser demand for reliability, simplifies the application's structure and implementation.

Reasoning about complex system may be simplified if an adequate structuring tools are used—we hope that the model of a group structure we propose may be used as such a tool in case of systems exploiting group communication. In the model, we emphasized functionality of components and abstracted of implementation issues. The model is suited for expressing structure of a group, interdependencies among its parts, and interactions with and within a group. The model may be customized to simpler group structures; more sophisticated groups may be described by combining multiple groups into an arbitrary interconnected structure.

We succeeded in adding the concept of group communication to an object-oriented language. The Multiple Object Invocation paradigm we devised may be used for express-

10.6 Summary of Implementation Issues

We have solved the following major implementation problems:

- broadcast communication is implemented by means of light-weight groups,
- broadcast protocols are used for implementing broadcast statement and group membership changes,
- members' invocations are accomplished by creation of new processes in the nodes members are residing in,
- each node maintains view of membership for all groups,
- only nodes in which member objects reside in have to know their location,
- mobility is orthogonal to broadcast invocations,
- broadcast operations are implemented by copying or sharing activation records.

Our solution is to change mobility to be reliable—either it succeeds or not. The last mobility message sent indicates that mobility succeeded. A source node removes the moved object from its local member list. A target node adds the object to its list. A subsequent crash of the target node is equivalent to failure or loss of the object. We implement a dynamic ascription: on move a member object identity is removed from the local table of members and added to the remote one. We do not force each node to know location of each member. Initially this information is available because subscription message is broadcast. Subsequent travels of an object are visible only foe involved nodes.

10.5.4 Monitor Operations in a Broadcast Statement

Different implementations were chosen for broadcast statements inside and outside a monitor. The non-monitored broadcast statements restrict access to variables and simplify implementation of broadcast operations. Activation records do not have to be copied back to a source activation record because no synchronization is necessary among broadcast operations. In contrast to that monitored broadcast statement do not restrict access to variables. Activation records have to be copied however to a main one, so that each next process executing a broadcast operation sees results of its predecessors' execution.

Entering and leaving broadcast operations of a broadcast statement inside a monitor is associated with entering and leaving a monitor. The process that initiates the broadcast statement leaves the monitor when it broadcast the invocation message. Processes that execute broadcast operations are treated as if they wanted to start a monitored operation. The monitor lock is checked and if the monitor is empty then they are executed. Otherwise, they are suspended. No guarantee is given that all processes executing a broadcast statement are executed before other processes are admitted into a monitor. Broadcast processes may be mixed with any other processes.

Monitored broadcast statements may have condition operations in them. The tradition semantics of Emerald conditions is preserved inside broadcast operations. A wait statement executed by a process inside a broadcast operation unblocks the monitor and suspends the process. Any other process, not necessarily inside the same broadcast statement, may enter the monitor. The signal statement causes a restart of any suspended process. Also, no priority is given to processes inside a broadcast statement.

An alternative approach could assume that signal and wait statements inside a broadcast statement concern only processes inside the statement. It could be useful because one could want the whole statement to be finished before unblocking other processes. This would lead to hierarchical monitor locks: one level for each level of broadcast statement nesting. We did not decide for this solution for a number of reasons. First, because it is too restrictive because it blocks other processes in a monitor until the broadcast statement is finished. Second, it may be simulated by conditions: monitor may be locked at the beginning of a broadcast statement execution, and unlocked at the end. Third, implementation of this approach would force a change in the Emerald objects—now there is only one lock per monitor, it would have to be replaced by a hierarchical one, or a process queue would have to be replaced by a hierarchical queue of queues.

for each such member so that when an invocation returns a stack for it is ready. If it was not ready then a returning process would have to be suspended. A check of for a ready stack would influence performance of a traditional remote object invocation because it would have to be performed on each return from a remote invocation.

Additional problem preventing us from preparing stacks for return from a remote invocation are the new processes's identifiers. They would have to be created before broadcast (which is impossible because it is not known how many of those processes has to be created) or a deterministic algorithm would have to be devised to assign identifiers after a number of member sis determined. This algorithm, on the basis of a number of member objects and their location at the message delivery time (it is detectable at each node so at the originating one too) would select an OID for those processes independently in the originating kernel and in all kernels invoking member objects. Even though such an algorithm is feasible (e.g., based on reserving OID numbers for processes created to handle broadcast) it would be expensive, additional messages would have to be exchanged and unnecessary dependency would be introduced - loss of synchrony among kernels would be disastrous. A cost of avoiding inconsistency plus cost of detecting it could also be substantial.

Yet another cost associated with central creation of processes comes from possibility of a node crash. If a process is created for a member object in a remote node and subsequently its kernel crashes then the member object will be detected unavailable and no additional cost will be suffered. If the process is created locally then an additional orphan stack frame is left for garbage collection.

Because of all those reasons, we decided to create processes in remote kernels and return the result values by different procedures then a traditional RPC.

10.5.3 Mobility vs. Group Communication

One of the distinguishing features of Emerald is its support for efficient fine-grained mobility. Addition of new primitives should always preserve semantics and performance of mobility. Especially, cost of mobility should not be increased, objects should be able to be move at any moment, and objects' invocation should be location independent. In case of our new constructs: broadcast should not delay mobility, objects executing a broadcast statement or being invoked via a channel should be mobile, and mobility should not influence the course of broadcast invocation. We do not affect efficiency of object mobility. No extra message is necessary to move an object.

There is a problem with synchronizing mobility with broadcast invocations. Until a message becomes deliverable it is impossible to say how many members is out there and where they are located. To prevent two node calling the same object some form of synchrony between mobility and broadcast invocation is necessary.

One of possible solution is rigid ascription of members to nodes where they subscribed. This is inefficient because if members are moved then the messages have to be forwarded. We could use extra broadcast message for making all nodes aware of new position of a member—this is unacceptable because we generate an extra broadcast message on each relocation.

10.5.1 Entering and Leaving Broadcast Operations

An initial process and a reply section of a broadcast statement are together referred to as broadcast operations. Implementation of broadcast operations is difficult because it requires admitting a number of processes to execute concurrently a code with shared data. A broadcast operation is executed in the following way. First, a new process and a stack segment for it are created. Second, an activation record for the operation is created on the stack. Third, the process is executed. When the execution is finished or a quit statement is executed then the process is killed and its stack segment discarded.

The main problem is with activation records: a number of different processes executes the same code and shares variables. There is no "logical" reason that would prevent a process executing a broadcast operation from accessing variables defined in the scope embedding the broadcast statement, so we have to admit it without violating semantics of Emerald and security of concurrent processes. It is impossible to implement reply sections and initial processes as Emerald operations because it would violate semantics of operations which may not be nested. We decided to treat broadcast operations as parts of a normal operation (the operation broadcast statement is placed in). They may be executed in parallel. We avoid in this way changes to semantics of an operation and yet permit access to all identifiers defined outside of the statement A constant **to** is defined in an imaginary block which encircles both an initial process and a reply section. If broadcast operations were treated as normal code then processes inside of them would have to be serialized. The reason is that no activation record exists in such a case and local variables for all those processes operate on the same stack mangling with each other data

Even if broadcast operations are not operations they could still be implemented as such. A compiler is able to create a definition of an operation based on our syntax. Their full implementation as operations would mean that a separate activation record is created for it. This in turn means that either a restriction is put on access to outside scopes or an implicit duplication of variables defined in outside (the whole activation record). The solution is a balance between implementing broadcast operations as implicit operations and serializing processes in broadcast operations. When a broadcast operation is started a process different from the broadcasting one and a stack frame for it, separate from the one on which broadcast was initiated, are created. Activation record is copied from the stack frame on which broadcast was started. In case of monitored broadcast statement activation records are copied back to the initial stack segment when a broadcast operation finishes. The reason is that different processes in the statement should see changes done by others.

10.5.2 Creation of Processes to Handle Broadcast Invocations

There is a choice between centralized and distributed creation of processes. In our case, a new stack was created remotely without complicity of the kernel in which broadcast was initiated. The originating kernel could create segments for all member objects but this would require a substantial synchronization at a time of return from a member object. Problem is that in such a case an originator of broadcast would have to create a segment

hand, information about which node invoked a particular member carries no information because a member object may have moved before finishing the invoked method.

Node failures detected by the STATE module are not signaled directly to a broadcasting process and they do not influence execution of a broadcast statement. If failures have to be detected and reacted to then a NodeEventHandler may be set. A failure of a node after a member has been invoked is equivalent to a failure if a Remote Procedure Call. Unfortunately, it is impossible to determine if a member object was invoked or not.

Information about collecting results could be used on the callee's side to save some messages. For example, if all replies are ignored (client resumes operations immediately after initiating broadcast) then no message could be send. If only 1 (k) replies are expected then only first 1 (k) from each node is send. This would require setting number of expected replies before initiation of the broadcast statement. We do not implement it because we focus on flexibility.

10.4.2 Returning Results

Results of a broadcast invocation are returned point-to-point. A sliding window reliable protocol is used. The Emerald traditional method of treating remote invocation results is applied in the case of results of broadcast.

10.4.3 Membership Messages

Messages changing the state of membership relation of any group is broadcast according to the atomic causal order. We require this strong ordering because atomicity is necessary because an object may be independently subscribed and unsubscribed at different nodes. If the messages were not ordered atomically then some nodes could order the messages differently causing inconsistencies in the membership. We require causality because if some changes in the membership relation for a group are done in a causal chain then implicitly a process which requires the changes assumes that the foregoing changes are effective.

In contrary to broadcast invocations, we employ a pessimistic approach to membership messages. We require stability of membership messages because inconsistencies at the application level would be very hard. We require also that messages are delivered only if the local state of receiving node is consistent with the view the message was sent in.

10.5 Other Implementation Issues

We have described how the basic broadcast and group operations are implemented. In this section, some other problems are described. We show here certain implementation decisions were made and how they influence the systems performance.

10.3.7 Reply Sections

When a message with results of a broadcast invocation is delivered to a kernel in which a client object resides then a reply section for this result is executed. A new process is created to execute the initial process. A new stack segment and an activation record is created for the process (see section 10.5.1). Results are copied from the message into the activation record and the process is started.

10.4 Use of Broadcast Protocols

Broadcast communication is used in the implementation of Multiple Object Invocation for two purposes: for broadcasting group membership information and for broadcasting MOI messages. In this section we show how broadcast protocols are used for those purposes.

10.4.1 Invocation Messages

MOI messages are broadcast according to the following protocol:

- messages are ordered by the atomic causal order protocol,
- messages are delivered in any state of a node, if the state is not consistent with the view the message was sent in then invocation in all member objects fails,
- stability is not required for a message to be delivered.

We use the atomic causal ordering because it is the strongest ordering and we want to provide the most transparent form of broadcast. Weaker forms would require some control of ordering at the level of Emerald programs. In the next versions of our primitives may be enhanced to support other orderings.

Communication failures should be visible at the level of the language because if unrecovered failures masked from a user of communication may cause inconsistent decisions. One of possible sources of inconsistency is message delivery in an unstable state after execution of the SSA procedure. Those inconsistencies are undetectable and hard to recover at the level of language. Therefore we decided to declare failures of invocations triggered in an unstable state.

We do not require stability of messages before their delivery for the sake of efficiency. According to an optimistic approach to failures, we prefer to deliver invocation messages when it is sure that the message will not be disordered. A node crash, which is the only reason of message not being eventually delivered, is an unrecoverable failure from the point of view of broadcast invocations. Therefore, it is better to deliver messages fast and risk blocking in cases of crashes. An application may set its own timeouts or trace placement of member objects to implement its own recovery policy.

A kernel initiating broadcast is not notified about invocations of member objects. It is unnecessary because the number of member objects which should be invoked is known (because subscription messages are ordered together with broadcast invocations). It is irrelevant which node invokes a member object as far as it is done only once. On the other

- GOID of a name of the operation being invoked, and
- arguments to the operation.

A message is marshaled identically to a traditional RPC message. It has a different message type then a traditional RPC message and a different handler is invoked after delivery of the message to a kernel. A message is broadcast over a network by the atomic causal protocol. When a message becomes deliverable, a handler for MOI broadcast messages is invoked. The handler unmarshals the message and calls a routine that triggers invocations in all member objects.

10.3.4 Member Objects Invocation

When a broadcast invocation message is delivered to a kernel then all member objects local to the kernel are invoked. A traditional object invocation is simulated by the kernel. First, a new process is created for the object and its identity remembered together with the identity of the process that initiated the broadcast statement. This information is used to locate a process to be unfrozen when a quit statement is executed. Second, an activation record for a member object is created in a way identical to the traditional remote object invocation, and arguments to the invocation are copied into the activation record. Third, an appropriate method of a member object is invoked.

10.3.5 Return from Broadcast Invocation

There is a return address at the bottom of each stack segment in Emerald which points to a procedure called when the bottom is reached. The procedure checks how the stack segment was created (e.g., for an object's process or to handle a remote invocation) and selects an appropriate handler.

When a method invoked via broadcast returns, a bottom of a stack is reached because a new stack segment is created for the invocation. A return from broadcast invocation is detected and a handler for this kind of returns is called. The handler creates a return message, marshals the results of the invocation into it, and passes the message point-to-point to the kernel that initiated broadcast.

The handler for return from a broadcast invocation performs identical actions as the one for remote invocations. The difference is in a different kind message being sent back and different methods of their handing after reception. A message for the traditional remote invocation triggers a return to an existing stack segment. A message for the broadcast invocation causes creation a new stack segment (see section 10.5.2) and execution a reply section on it.

10.3.6 Initial Process

After an invocation message becomes deliverable to a kernel in which a client object resides a procedure is called to start an initial process in the client object A new process is created to execute the initial process. A new stack segment and an activation record is created (see section 10.5.1) and the process is started.

1. A kernel is called with arguments to the invocation and identity of a channel and the method being invoked.

2. The kernel constructs an invocation message from this information and broadcasts it to all kernels in the system.

3. The kernel creates a process to handle an initial process of the broadcast statement.

4. When the invocation message is delivered to a kernel then a process is created for each member local to the kernel and the method is invoked on it.

5. Upon return from the method, results are returned to the kernel that initiated the broadcast statement.

6. For each returned result a new version of a process is created and a reply section is executed.

10.3.2 Initiation of Broadcast

Initiation of a broadcast statement is implemented by a kernel call. This call takes the following parameters: identity of an open channel used in the statement, identity of the method being invoked, and arguments to the invocation.

Code generated for an initiation of a broadcast statement simulates a traditional object invocation. A stack frame for a broadcast invocation is built like in case of a traditional invocation:space for results is reserved, arguments to a broadcast invocation are evaluated and put on a stack. A pointer to an open channel, a pointer to its *AbCon*, and the identifier of an operation being invoked are stored in registers. The code is generated as if the open channel object was being invoked with the method to be broadcast. Instead of calling the object, a kernel is trapped to. In a traditional object invocation, pointer to a code of a method being invoked is linked into invocation code at the loading time and an object may be called directly without kernel's complicity. It is impossible to call member objects directly because broadcast invocation is always a remote operation which is performed by a kernel.

When a stack frame is created then a kernel is called. The kernel turns a "cheated" traditional invocation into operating system actions leading to invoking all member objects in the system. The process executing a broadcast statement is preempted and it waits until the invocation message broadcast on behalf of the process becomes deliverable in the current node of the process. A broadcast invocation message is created and broadcast.

10.3.3 Broadcast of Multiple Object Invocation Message

A complete MOI message consists of:

- GOID of the open channel used in a broadcast statement,

- GOID of the process executing the broadcast statement,

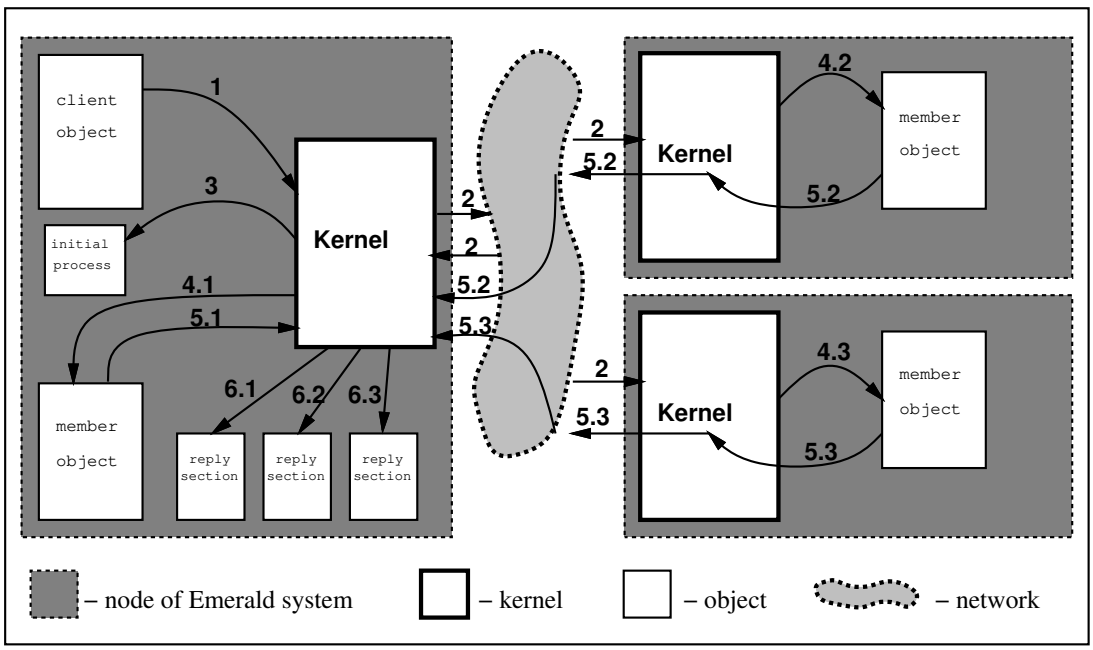

Figure 10.1: Scenario of a Broadcast Statement

group membership it conveys are introduced locally. Consistency among each node's view of membership is assured by broadcast protocols.

A list of member objects for an open channel may be obtained locally in a node without communicating with others. When the list is requested then the local representation of the global membership of a group is returned. It is unnecessary to synchronize such a request with messages changing membership because the messages are delivered in an orderly manner and at each moment in time a node has a consistent view of membership.

10.3 Implementation of a Broadcast Statement

In this section, a detailed discussion of an implementation of operations performed during a broadcast invocation is given. First, an execution phases of a broadcast statement are shortly presented. Next, each step is in detail discussed, implementation problems are pin-pointed and rationale for our solutions are given.

10.3.1 Scenario of a Broadcast Invocation

Execution of a broadcast statement consists of six phases (Figure 10.1): initiation of broadcast invocation, broadcast of an invocation message, creation of an initial process, invocation of member objects, returning results, and collecting results:

appropriate protocols. Also, those of the new language capabilities that require support from a kernel, have to be defined in terms of operating system notions like process or activation record, and implemented in the system. Reliability of the channel operations and broadcast invocation are based on the paradigm of virtual reliability.

10.2 Channels and Channel Operations

The new Emerald group mechanism is based on the *Channel* built-in object. A group in the language is represented by an open channel. Channel operations include creation of an open channel (and thus a new group), subscribing and unsubscribing an open channel (joining and leaving a group).

10.2.1 Channels Related Information

An open channel (a group representative in the language) is represented in the system by its GOID (Global Object Identifier). This assures that each group has a distinct identifier in the whole network independent of its creation site. GOID of a channel is passed in broadcast invocation messages. It is used for finding a set of current member objects of the group it represents.

A type of a group (i.e., a type associated with an open channel) is not represented explicitly in the system. The compiler assures that all objects' operations are type-safe. Each object is represented in the system by its identifier (GOID or ODP) and a translation table (AbCon). This information is enough to perform all channel operations and a broadcast statement.

A group consists of member objects. Each node of a system maintains information about all groups existing in the system. The group information in a node consists of two lists: a member list containing identities of all member objects of the group, and an additional list of those member objects that reside in the node. The general member list keeps GOIDs of member objects and represents a current state of the group membership. The local member list contains members' OIDs.

10.2.2 Organization of a Group

There are four channel operations: creation of an open channel, subscribing to a channel, unsubscribing to a channel, and obtaining a list of objects that subscribed to a channel.

Creation of an open channel is merely language-level operation. A new object (the open channel) is created locally in the creating node. The open channel object is immutable which means that it is replicated in all nodes it is referenced. The object is not replicated immediately after creation in all nodes. Instead, its replication is postponed until it is explicitly invoked (in any of the channel operations).

Operations of subscribing and unsubscribing to an open channel are treated uniformly. When an object invokes such an operation on a channel then a message describing this operations is broadcast. When the message is delivered to a node, the changes to the

The Emerald project was initially aimed at creating an efficient system supporting objects. This aim should not be violated by the extensions we propose: new operations should be efficient, and their addition should not influence efficiency of existing mechanism. Our aim is not to influence existing applications. New mechanisms should be "invisible" to a programmer not using them—no significant change in performance should be noticed. Our mechanisms are extensively using network communication therefore we are specially concerned about those mechanism that are dependent on networking: remote invocations and mobility.

10.1.2 Light-weight Groups in Emerald

Light-weight groups are groups comprised of light-weight entities. Light-weight objects are used in the case of Emerald; light-weight processes could be used in non object-based systems. Creation of a system based on light-weight components poses different problems then a heavy-weight system.

First of all, light-weight entities are in most cases encapsulated in their heavy-weight counterparts, e.g., light-weight processes are implemented inside heavy-weight ones. This creates a situation in which a number of light entities have to be supported by one heavy entity. For example, a grain of communication in Unix is a socket which belongs to a single heavy-weight Unix process. If a number of light-weight processes inside a Unix process take part in communication then either messages for them have to be multiplex into one socket or a socket for each has to be created. The situation is even more complex if the light-weight processes take part in group communication: either messages are duplicated before sending or a receiving heavy-weight process duplicates them before delivering them to light-weight destinations.

The second problem created by lightness of Emerald groups is fine-grained mobility. Member object may move from a node to another changing membership in nodes. As messages are passes to nodes which forward them to objects it is necessary that nodes know exactly which of them should pass a broadcast invocation to an object. This problem does not exist in case of simple invocation because there is a unique invocation message which is forwarded to its destination object. This forces some form of synchronization of mobility and group communication. Full synchronism among all nodes is unacceptable because it would degrade mobility performance. Synchronization between the two nodes taking part in mobility is, however, enough and may be done at no cost.

10.1.3 Implementation Issues

Implementation of channel-based communication consists of three main parts: implementation of channel operations, implementation of the broadcast statement and assuring reliability of the two.

Implementation of channel operation consists of implementing creation of channels and of operations changing group membership. The main issue in implementing the broadcast statement is use of the new interprocess communication module. Each step of the sentence has to be translated into sequences of messages passed to remote nodes by

Chapter 10

Implementation of Light-Weight Groups

Chapter ?? presented the idea of Multiple Object Invocation as a method of accomplishing group communication among objects. In the fifth chapter, Emerald realization of MOI by means of channels and the broadcast statement was proposed. This chapter shows how channels-based Multiple Object Invocation is implemented by means of light-weight groups.

10.1 Overview of Implementation

In this section we give an outline of the implementation of the Emerald's group communication primitives. We have based the implementation on light-weight and fine-grained groups. The groups are light-weight because processes involved in the group communication are light-weight. The groups are fine-grained because Emerald objects, which may be of arbitrarily small size, are members and clients of groups.

10.1.1 Goals and Properties of the Implementation

The goal of adding group mechanisms to the Emerald system was implementing the new language constructs facilitating group communication. Two kinds of operations have been added to the system: first, group mechanisms corresponding to the channel operations, and second, Multiple Object Invocation mechanism implementing the broadcast statement.

Introduction of new constructs to a language should not degrade performance and change semantics of the language. In case of the Emerald group communication primitives, we want neither any additional restriction imposed on the language, nor any change of the meaning of already defined features. Such a smooth extension assures that every program that could be run on the previous version of Emerald will be able to run on a new one. Even though the number of Emerald users is limited those who could be interested in upgrading their system and language to the new version should not experience any troubles. Especially, we want to assure that no porting of programs is necessary.

9.5 Summary of the IPC Module Features

The new interprocess communication module that is described in this chapter exhibits a number of interesting features:

- it implements broadcast and point-to-point communication,
- it provides various methods of ordering,
- it provides different levels of reliability,
- it may be used in the virtual reliability approach.

LH module

The module remembers the time of reception of the last message from each node. It is used to toggle SSA procedure if no message was received from some nodes for a long time.

ES module

The module maintains a list of system's views that are consistent with the current local view.

9.4.3 Initiation of the SSA Procedure

The SSA procedure is triggered whenever inconsistencies are detected in the system (see section 8.3). There is a number of additional situations in which the SSA procedure is also started:

- An observed lack of communication may be caused by any of three reasons: the system may be idle, other then the observing nodes might have crashed, the observing node is separated (a partition occurred). The goal of calling SSA is to check what is the actual state of the system. In the first case, the SSA procedure confirms presence of all nodes, in the following two cases, the local view is changed to reflect observed system state.

  Triggering the SSA procedure because of lack of communication causes an idle system to exchange messages. Such a periodical exchange of broadcast messages is unnecessary if the system is idle for a long time. A back-off scheme for initializing the SSA procedure is used to save unnecessary communication. The back-off scheme causes a kernel that does not receive any message to trigger subsequent invocations of the SSA procedure with increasing intervals.

- If no message has been received from a node for a long time then there is a suspicion that the node is down. A *COMMIT_LIST* message is reliably sent point-to-point to the suspect node. This message forces the suspect node to acknowledge it. If the node does not respond then the SSA procedure is started because of unacknowledged point-to-point message.

- If a point-to-point message is not acknowledged for a long time in spite of retransmissions then the SSA procedure is initiated to either exclude the faulty node from the system or force it to reply.

- If too many messages have been received in the BROAD's buffers and not delivered then the SSA procedure is invoked to force other nodes to flush their buffers. This primitive form of flow control assures that the buffers may not be overflowed.

- After the bootstrap procedure is successfully finished the booting node starts the SSA procedure to announce its existence.

messages from nodes not in the current system view if a message is received from a node not in the current local system view then actions leading to incorporating the alien node into the view have to be undertaken. The FILTER module passes the message to the SYSTEM module which performs those actions.

messages sent in an unknown view if a message is received from a known node but it was sent in a view not included in the local list of equivalent states then it is passed to the SYSTEM module which recovers from potential inconsistencies.

Messages that remain after filtering come from known sources that are in a state consistent with the receiving kernel's view of the system. Additionally, in the BOOTING state, the FILTER module drops all messages except for *COMMIT_LIST*.

9.4.2 Structure of the SYSTEM Module

The SYSTEM module consists of a number of submodules which implement different parts of the module's functionality. The **SSA** module implements the System State Agreement procedure. The **BOOT** implements bootstrap procedure. The **NL**, **COL**, and **ES** modules keep information necessary for correct operations of the SYSTEM and FILTER modules.

SSA module

The SSA module implements the System State Agreement procedure. It may be viewed as a state machine with state corresponding to phases of execution of the procedure, and evens equivalent to messages received.

BOOT module

The BOOT module implements the bootstrap procedure. At the beginning of the procedure the module broadcast the *BOOTSTRAP* message. Subsequently, the booting kernel ignores all messages except for *COMMIT_LIST* messages. Each accepted message causes the kernel to update its node lists. If no duplicate node with the same *LNN* is detected then the kernel starts its normal operations and triggers the SSA procedure.

NL module

This module keeps a list of active nodes in the current state. The list is the kernel's current local view of the global system's state. It is used for filtering messages coming from unknown sources.

COL module

This module maintains a list of nodes used during the COLLECTING phase of the SSA procedure. The list contains identity of known nodes and the confirmation flags for them.

- each message is marked by the sending node's view of a global state. A message that is requested to be delivered only in the same view is discarded if it becomes deliverable in a view that is not equivalent to the view the message was sent in,

- a message that is requested to be delivered only in a stable state is discarded if it becomes deliverable in an unstable phase of post-SSA operations,

- messages that may be delivered in any local view and without concern about stability of delivery is never discarded.

9.4 Consistency Maintenance Protocol

Consistency maintenance is a crucial part of the new IPC module. It makes reliable broadcast feasible and simplifies applications which do not have to check consistency by themselves.

9.4.1 FILTER Module

The FILTER module assures that a kernel does not receive messages from nodes not in the current local view of the system. The filtering is performed in four phases in which different kinds of messages are filtered-out:

messages from a node with the same LNN if there exist two nodes with the same *LNN* then one of them has to be killed. Therefore:

- if a message is received from a younger incarnation, a *COMMIT_LIST* message is sent which will make the other kernel abort further operations,
- if a message is received from an older incarnation, the receiver stops its operations.[2]

No concern is given to the case in which two different nodes have the same *LNN* number and the same incarnation time. The probability of an event of two kernels setting their incarnation time to the same value is negligible low. Checking if this unlikely case occurred would require checking address of a sender of each message coming with the *LNN* and incarnation time of the receiving node. This would add an unacceptable overhead to processing of own broadcast by a node.

SSA and BOOT messages SSA and BOOT messages are accepted by the SYSTEM module regardless of the current state of the receiving kernel and its local view of the system. These messages may not be passed to the BROAD module because they could cause inconsistencies. The FILTER module detects BOOT and SSA messages and calls appropriate handlers for them.

[2] In the current implementation, the stopping kernel commits a suicide by sending the *KILL* signal to itself.

record. When the selected message is delivered, delivery conditions for all messages for which backward links exist from it are checked. If the delivery conditions are still not fulfilled then a new message is selected for each dependent message.

atomic order – an ORD record for an atomically ordered message contains a link to the sequencer's message which orders it. When the first sequencing message is received then the record for the first message in the atomic order is marked as deliverable. It is delivered immediately if it has been received. Otherwise, it is delivered when it is received. After a message ordered atomically is delivered, the sequencer's message pointed to by the delivered message's link is examined. All messages it acknowledges are checked in turn. The first one to be ordered atomically is marked as deliverable. If no such message is found then the next sequencer's message is checked. If there is no sequencer message that could point to the next message in the atomic order then ORD remembers that the next sequencing message should be treated as the first one.

atomic causal order – first, the causal dependencies have to be fulfills for a message sent in this order. This is achieved identically as in the case of causal order. The atomicity constraints are fulfilled when a message which is concurrent or dependent on a given one is received from each node. The ORD maintains a table in which next messages received in order from all nodes are remembered. When the table contains a message from each node then the messages in the table are checked. All those which acknowledge only already received messages are sorted and delivered. The other messages remain in the table for the next cycle of the algorithm.

9.3.5 Message Stability

Message stability is used for assuring that messages are delivered only if a certain number of nodes receive them. Message stability may be used for implementing approximate atomicity of broadcast. Also, if lesser reliability conditions are required then lower level of stability allows more efficient operations due to reduction of buffering time for messages.

Message stability is checked on the basis of acknowledgments. For each message the ORD keeps record of all nodes that acknowledged the message. When a new acknowledgment for a message comes then the delivery conditions are checked for the message. In the required level of stability (number of nodes that have acknowledged the message) is reached then the message is delivered.

9.3.6 Consistency Requirements

Consistency requirements determine what is the acceptable level of consistency between the state of the sending node in the moment of sending and the receiving node in the moment of delivery. If the conditions are not fulfilled then a message is treated as undeliverable and is discarded.

The three admissible consistency requirements have their implementation conditions that are equivalent to the requirements:

9.3.2 Examples of Delivery Specification

- no constraints (ORD = *NO*, STAB = 0, CONS = *NO*) a message is delivered and discarded immediately after reception
- reliable, causal - (ORD = *CAUSAL*, STAB = *ALL*, CONS = *NO*) a message is delivered according to causal order, it is kept by a process until all processes receive it.
- ordered, 3-stable, same state - (ORD = *ATOM*, STAB = 3, CONS = *STABLE*) a message is delivered in an orderly fashion, only if at least three processes are known to have received it and only if the system state is stable.
- atomic causal order, stable, same state - maximal constraints (ORD = *ATOM-CAUSAL*, STAB = *ALL*, CONS = *SAME*) a message is delivered according to atomic-causal ordering, only if all processes have received it (atomically) and only if the system state is preserved.

9.3.3 Partial Order Graph

The ORD module maintains a graph which reflects a partial order induced over messages by acknowledgments. For each message there is a record created which holds information about the message relevant for ordering protocols. Those records are put into a graph. Nodes of the graph are messages's records while edges are acknowledgments they piggy-back. When a message acknowledges another message then the fact of acknowledging is notified in the record for the message being acknowledged. If the acknowledgment is positive then it is propagated down the chain of positive acknowledgments until a message already acknowledged by the acknowledging node is met or no more positive acknowledgments is found. When an acknowledgment is added to a message's record, the message delivery condition is checked If the condition is fulfilled then the message is passed to appropriate message handlers.

If a not received message is being acknowledged then an empty record is created for it. All ordering information is gathered in this record but is not processed and not propagated until the message is not received. When the message is received then the accumulated acknowledgments are processed and passed down the chain.

9.3.4 Message Ordering

Messages are ordered solely on the basis of acknowledgments:

no constraints – when a message is received, a record for it is created in the graph of partial order, and the message is immediately delivered.

causal order – a message is delivered when all messages it directly depends of have been delivered When a message is received and the dependent messages have been delivered then the message is delivered. Otherwise, one of the undelivered messages is selected and a backward link is created from its record to the dependent message's

message for long time), the BROAD module uses outgoing non-BROAD messages to piggyback acknowledgments.

An advantage of using non-BROAD messages instead of dummy messages is increased efficiency. Network traffic is not increased because the used message was to be sent anyway. Number of interrupts is reduced by 1 or n because a dummy and a non-BROAD message would trigger $n+1$ (if the non-BROAD message was sent point-to-point) or $2n$ (in case of broadcast) interrupts. A special BROAD message type is created for the non-BROAD messages broadcast by BROAD. A default handler for this type discards messages not destined originally for the receiving node, and passes messages to appropriate upper-level handlers otherwise. Messages of this type are broadcast by BROAD without any ordering constraints so their delivery is never delayed.

Because of using non-BROAD messages for piggybacking BROAD acknowledgments, two protocols for one message are used. There is no problem in case of unreliable point-to-point and broadcast message passing. The sliding window messages, however, become object of double acknowledging and retransmissions. Double acknowledging is necessary because neither of the two suffices. The sliding window acknowledgments are valid only between two nodes and may no be used for broadcast ordering. The BROAD acknowledgments should not be used in the sliding window protocol because the subsequent sliding window messages would become dependent on BROAD messages. This could cause delays in their delivery.

9.3 Ordering and Delivery Protocols

Ordering and delivery protocols are implemented by the ORD module. The module constructs a graph reflecting the partial order induced over messages by acknowledgments they piggyback. When a new message is received, its acknowledgments are processed and if any message becomes deliverable then it is passed to appropriate handlers.

9.3.1 Different Levels of Service

A number of different delivery requirements may be specified for a message. The first requirement specifies which ordering protocol should be used for a message. The protocols assure that a message

A parameter for each class of requirements is specified for a message:

type of ordering : no ordering, causal order, atomic order, causal atomic order

stability : a message may be unstable, k-stable, and fully stable,

consistency : a message may be delivered in the same view of the system state it was sent, in any stable view of the state, or without taking the system's state into account.

interrupt handling routines. The construction of a message from the highest levels (the LM module) to its sending is performed as one action of a kernel. Prolonging this action by additional processing associated with buffering and ordering leads to increasing waiting time for other, possibly urgent, interrupts. The message is anyway added to the TIM queue before sending so it is not lost even in case of the Unix network routines' failure.

The TIM module assures that kernel's own messages are not lost even if no node receives it initially. The module maintains a queue of not acknowledged messages and has a timer which triggers retransmissions. There is a counter of retransmissions for each message in the queue. A retransmission flag maintained by the module indicates that no acknowledgment has been received for own messages.

The TIM module works according to the following algorithm:

- Each new own message is added to the queue right before sending. Counter of retransmissions for this message is set to zero. If the timer is not active it is set for a new period and a retransmission flag is set.
- When any of the incoming messages acknowledges positively any of the messages in the queue, the retransmission flag is reset and the message removed from the queue.
- When the timer expires it checks the retransmission flag.
- If the retransmission flag is is reset then if the queue is empty the timer is deactivated otherwise the retransmission flag is set and the timer started for a new period.
- If the retransmission flag is set then the first message in the queue is checked.
- If its counter of retransmissions is bigger then the maximal number of retransmissions then the break in communication is detected (we have a situation in which outgoing messages are not received even though there may be some incoming ones successfully received). The SSA procedure is started to either recover from the situation or to declare a change of the system state.
- If the retransmission counter if lower then the maximum then the first message in the queue is retransmitted, the counter incremented and the timer set for a new period.

As it was mentioned before, it is crucial that the TIM's operations are cheap. The key is optimization of queue operations because they are performed for all own broadcast messages and acknowledgments for them. First, adding a message to a queue is done by two assignments. Second, messages are not removed from the queue when acknowledged but instead an extra queue of removed messages exists. The physical removal is done when the timer expires.

9.2.6 Point-to-point Massages

All messages sent out by an Emerald kernel (BROAD and non-BROAD messages) are passed to the network via the BROAD module. If it is necessary (no outgoing BROAD

- if there is an outgoing broadcast message it takes all accumulated acknowledgments with it. It is guaranteed that acknowledgements will fit into it.
- the acknowledgment queues are checked periodically and if there are pending acknowledgments and there was no outgoing message for a long time then a dummy message is sent to piggyback the acknowledgments.

An alternative solution could postpone sending a dummy message until an outgoing message is available. In this solution overflow is not checked when a message is received. Instead, when a message is broadcast and acknowledgments do not fit into it then a sufficient number of dummy messages is created and sent before the actual message. An advantage of this solution is that the cost of acknowledgment processing is reduced. A disadvantage is that messages are created in bursts—a very unwanted feature because bursty communication increases likelihood of omissions.

The two alternative strategies for adding acknowledgments may be also applied in case of a floating length of the acknowledgment field of the message header. The restriction about the fixed length of the field has to be replaced by a maximum number of acknowledgments that fit into one UDP packet sent over Ethernet.

9.2.4 Retransmissions

The current retransmission policy is to use broadcast for retransmissions and to differentiate between own messages and messages received from other nodes. Node's own messages are retransmitted immediately while others are postponed for a random period of time. The purpose of this postponing is to reduce likelihood of duplicated retransmissions.

The RET module has a queue of messages waiting for retransmission and a timer which triggers retransmissions. The module works according to the following algorithm:

- Whenever a request for retransmission is detected (a negative acknowledgment is received) the requested message's identity is passed to RET.
- If the request if for own message then it is retransmitted immediately, otherwise it is added to the RET queue.
- If the message is added to an empty queue then a timer is started for a random period of time.
- When the timer expires, the first message in the queue is taken (if there is any—they might have all been removed) and passed to BRD for retransmition. If any message is left in the queue, the timer is set for a new random period.

9.2.5 Processing of Own Messages

Kernel's own messages are treated uniformly with other messages. A message is placed in buffers and ordered by its source node after the node receives it. We do not process node's own messages at the stage of sending because messages are delivered reliably to the original sender and it is profitable to split the action of sending a message into two

acknowledgments – each acknowledgment consists of a node number and a sequence number and a flag indicating if the acknowledgment is positive or negative; a number of attached acknowledgments is also contained in a message,

view identifier – consists of identity of the node that declared the view and a sequence number of the view.

message type – determines which handlers should receive a message upon reception.

retransmission header – a message may be retransmitted by any node, to identify the actual sender of a message, its number and incarnation time is added to a message.

Number of acknowledgments attached to a message may vary so the acknowledgment part of the message header may have a different length. In the current implementation, we force the acknowledgment part to have a fixed length. Extra messages are to carry acknowledgments that do not fit into the fixed field. The fixed length solution wastes bandwidth in cases when less then the maximum number of acknowledgments is being carried. We have decided to adopt this method for its simplicity. The floating length header may be used effectively if a total length of a message may have a varying length. Emerald uses Ethernet packets for its communication. The packets have a fixed maximal length so messages with different header lengths would have different length of the message body field. This would force a higher level protocols to ask BROAD for an expected length of the acknowledgment field and to adopt length of each of their messages.

9.2.3 Acknowledging Policy

The ACK module maintains three queues of acknowledgments: the main queues for outgoing positive and negative acknowledgments, and an auxiliary one for those negative acknowledgments that have been sent but the message they acknowledge have not been received yet.

When an acknowledgment is passed for sending, it is added to an appropriate queue for outgoing acknowledgments. After they are piggybacked, positive acknoweldegments are removed from the positive acknowledgment queue and negative acknowledgments are moved from the negative acknowledgment queue to the auxiliary queue. A negative acknowledgment is removed from the auxiliary queue after the message it acknowledges is received. Negative acknowledgments are retained in the auxiliary queue to avoid sending duplicate requests for retransmission.

The acknowledging policy currently implemented is based on the following rules:

- a fixed number of acknowledments may be added to a message,
- acknowledgements are collected in appropriate queues,
- when a message comes and the acknowledgments it carries cause an overflow of acknowledgements (i.e., the next message wouldn't be able to to carry all of them) then a dummy message is constructed to flush the accumulated acknowledgments,

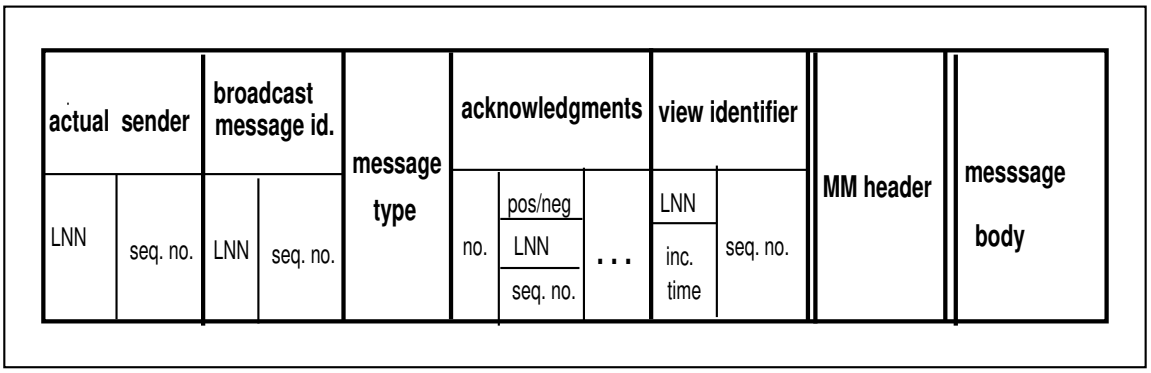

Figure 9.4: Format of a BROAD message

nodes even if no node initially receives them. If a message is broadcast and not received by any node then the omission may be detected only when the next message is broadcast from the same source. In cases when no message is broadcast, retransmissions are necessary until any node acknowledges the message. The type of failures that is detected by the TIM module is very rare and its operations are performed for each outgoing broadcast message. Therefore the costs of the module's operations should be negligible not to decrease the overall efficiency.

9.2.2 Message Format

Each message is identified by a sending node identity (see **??**) and a sequence number. Each kernel counts outgoing broadcast messages and adds the current contents of a count register to all broadcast messages. There is no synchronization among sequence numbers of messages coming from different incarnations of the same kernel. This synchronization is not necessary because each kernel keeps track of incarnation times of all known nodes and compares them with the incarnation time of a kernel issuing a message. In this way obsolete messages may be discarded and inconsistencies detected.

An outgoing message is created in two phases: first, the message is passed to the MM module which adds its header, second, the message is passed to the BROAD module which adds broadcast-related information. The structure of an Emerald message may be seen in Figure **??**.

message body – is the original contents of the message.

MM header – is the header added by the MM module. It includes the sliding window protocol information like, for example, destination address, source address, sequence number.

BROAD header – contains information necessary for the transport and ordering broadcast protocols. Ii includes:

broadcast message identifier – consists of a sending node's *LNN* number and a message sequence number,

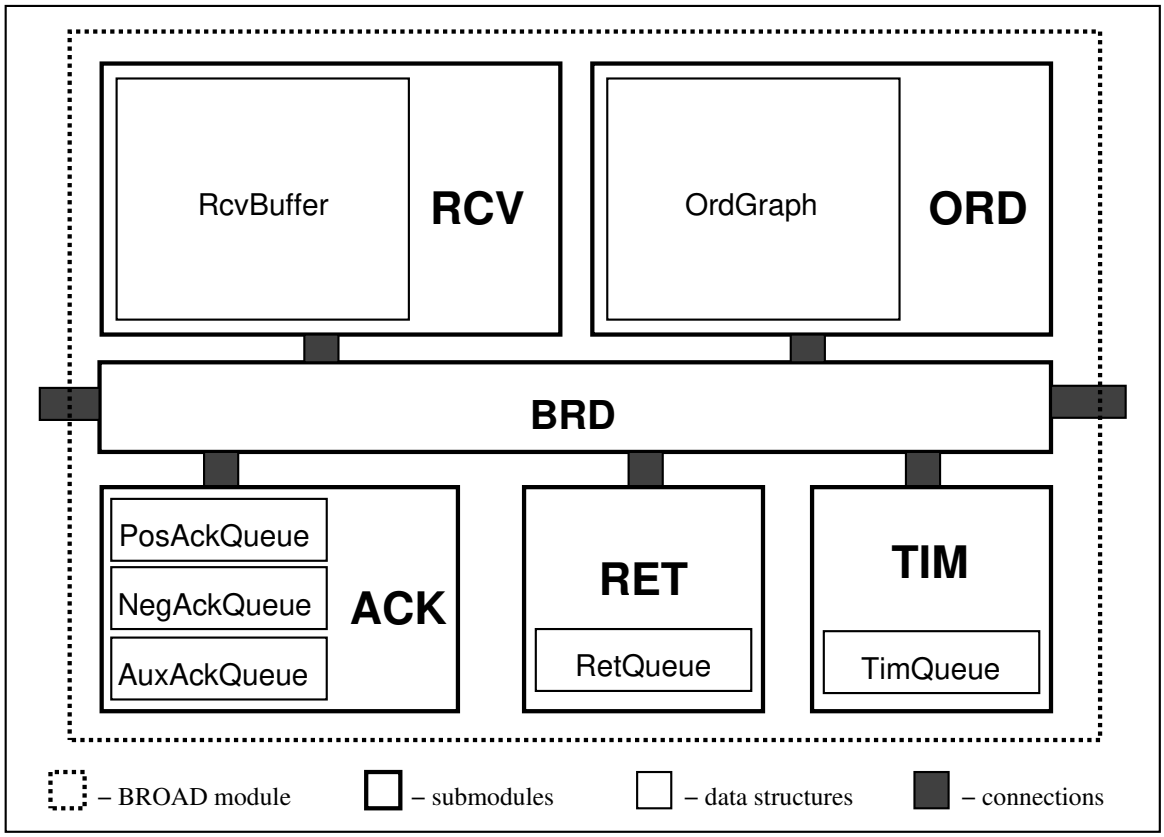

Figure 9.3: Structure of the BROAD module

- negative acknowledgments are kept available (it is possible to check if they have been passed to the module) after adding them to a message until they are explicitly removed,
- positive acknowledgments are canceled (forgotten) automatically after adding to a message.

RET module

The **RET** module maintains a queue of messages waiting for retransmission. The purpose of the module is to encapsulate message retransmission policy. It has two entries: one for requesting a retransmission of a message and one for canceling retransmissions. If a request for message retransmission is passed to RET then the message is eventually retransmitted unless the request is canceled before the retransmission occurs.

TIM module

The TIM module maintains a queue of the node's own messages that have not been acknowledged yet. TIM module assures that node's own messages are delivered to other

its receivers. A receiver registers for certain types of messages and subsequently receives messages of the specified types. If multiple handlers subscribed for a message type then messages of this type are duplicated and delivered to all handlers that subscribed for the type.

9.2.1 Structure of the BROAD Module

The BROAD module is built around data structures used for storing messages and protocols related information (Figure 9.3). The functionality of the transport and ordering protocols is split between RCV and ORD. RCV detects omissions on the basis of gaps between messages coming from the same source, while ORD detects omissions on the bases of acknowledgments, orders messages and detects delivery readiness of messages.

BRD module

The BRD module is a simple traffic organizer for messages: it passes messages to and from the BROAD as well as among its parts. BRD is called-up by a network interrupt handling routine when a new message is received. It places the message in the buffers (RCV) and in the partial order (ORD). It is also called to send a message over a network: a broadcast header and acknowledgments are added, and the message is sent.

RCV module

The RCV module is a main receptacle for messages. It provides a functionality of a typical buffer—messages may be added, removed, their presence in the buffer may be checked. All received messages are kept in the RCV buffers until they may be discarded.

ORD module

This module is responsible for messages ordering, detection of omissions, and detection of delivery requirements fulfillment. The module maintains a graph which is the node's current view of a partial order put on the messages on the basis of acknowledgements.

ACK module

This module accumulates acknowledgments and adds them to the outgoing messages. The module maintains buffers in which acknowledgments are stored before they are added to outgoing messages. The ACK module may implement different acknowledgment policies as far as the following conditions are fulfilled:

- all acknowledgments passed to the module and not removed subsequently are sent not later then with the next broadcast message (ACK may force creation of a dummy message to piggyback acknowledgments at any time),
- addition of an acknowledgment to a queue is idempotent if the acknowledgment is not removed from the queue in the mean time,

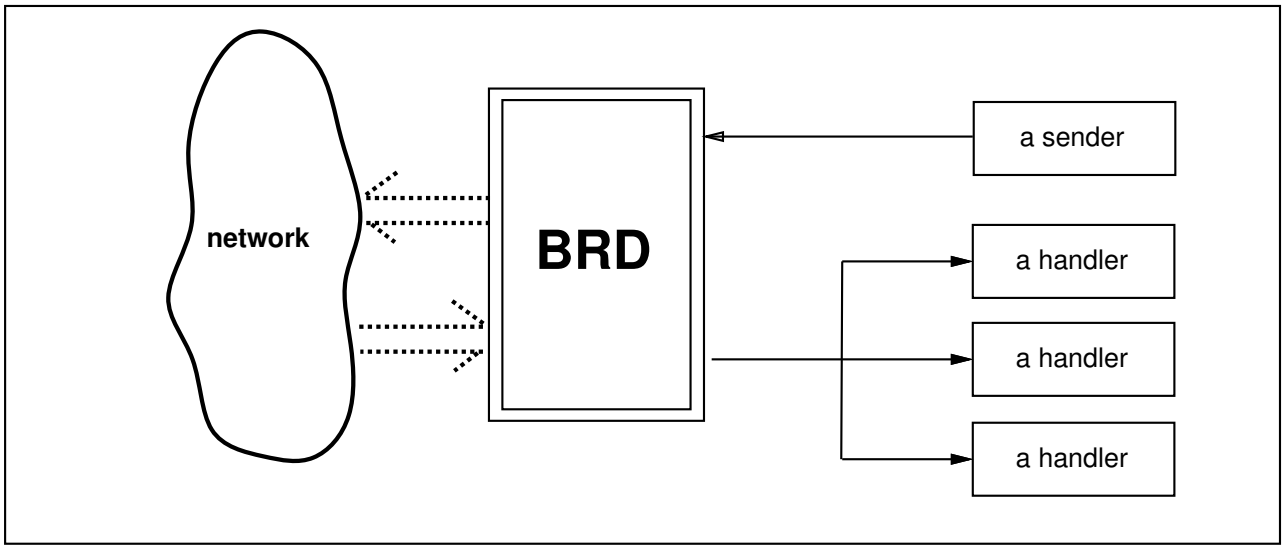

Figure 9.2: BROAD module

of a kernel may request system's state related information from the SYSTEM module (10). This information may be returned (11) immediately, periodically, or when the system's state changes.

9.2 Broadcast Protocols

The transport protocol and the ordering protocols are so tightly intertwined that that they constitute a single block with a simple functionality. Combined, they may be viewed as one module which provides two kinds of services:

- sending messages over a network, and
- delivering messages received from a network

The first kind of service is accomplished by passing a message to be sent, together with requirements for its delivery to the module. The second kind of service may be obtained by adding a handler to the current set of handlers. A handler is invoked and passed messages when they are ready for delivery.

In the Figure 9.2, the BROAD module and its logical connections are shown. From the point of view of the module's user, BROAD is connected directly to a network. A user sees a virtual system which consists only of known nodes. The FILTER module assures that messages sent from nodes not in the virtual system are not received by BROAD and thus are invisible to the modules users.

There are two kinds of client modules: senders and receivers. A sending module accesses the BROAD by passing a message to it. Each message has a type which determines

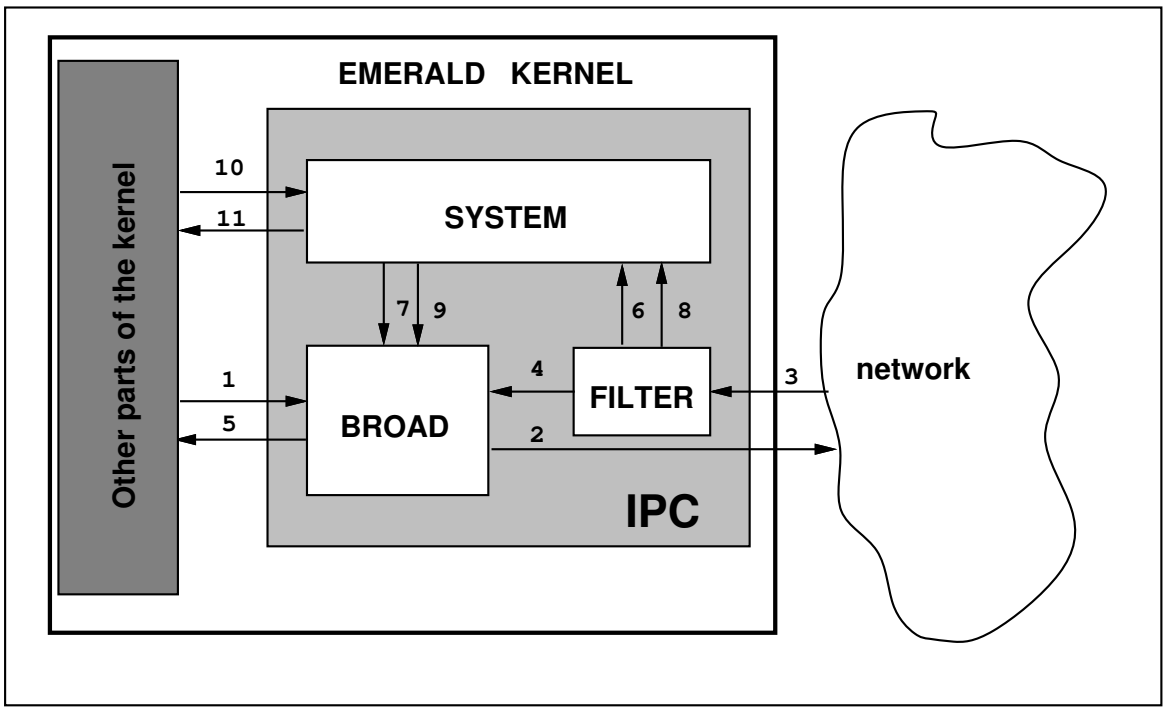

Figure 9.1: Position of the new IPC module in the Structure of the Emerald System

of the Emerald kernel see the module as two subsystems: one for control of the system state (e.g., information about the current state may be obtained fro SYSTEM), and one for network communication (sending and receiving messages via BROAD).

The previously existing communication routines are placed outside the new IPC module. They generate higher level messages which are passed to a network by the IPC module. For a more detailed discussion of the cooperation between the new IPC module and higher level protocols refer to the section 9.2.6.

Let us have a look at work of the IPC module. Messages that are passed to the module for sending over a network (1) are send by the BROAD module over a network (2). Messages that are received from a network (3) are delivered to the FILTER module.[1] The FILTER module checks if a received message comes from a node that is known and that the sending node's state is consistent with the receiver's state. If it is a case, the message is passed to the BROAD (4) which orders it and delivers (5) to the appropriate part of a kernel. If a received message indicates that inconsistency may have occurred in a system then the message is passed to the SYSTEM module (6). During its operations, the SYSTEM may send (7) and receive (8) messages. SYSTEM sends messages via the BROAD module but it receives them directly from FILTER. Some messages are also passed by SYSTEM to BROAD for ordering and delivery without sending (9). Other parts

[1]On the figure, we have omitted interrupt handling routines which receive message directly from Unix network routines and pass them to the FILTER module.

delivered in a causal order are still deliverable. Even in case of detecting inconsistency in a system, the point-to-point sliding window protocol works correctly. Independence of protocols is a great advantage because parts of a system that do not depend on components influenced by failures may continue their operations even if other modules are blocked.

modularity — the interprocess communication system consists of a number of modules implementing various protocols and structured by means of upcalls[Clark 85]. Each module may be replaced by a new one implementing the same functionality

9.1.3 Functionality of the IPC Module

Functionality of the new communication module may be split into two groups: the first one, implementing communication protocols, and the second one, assuring reliability and maintaining consistency.

The module implements the protocols presented in the seventh chapter: the transport protocol, and the four ordering protocols (unordered, causal, atomic, atomic causal). The module implements datagram communication, i.e., each message it sends is treated as a self-contained unit. Delivery requirements are set for for each message separately. The requirements include: an ordering protocol to be used, level of stability that has to be achieved to enable delivery, and consistency prerequisites that have to be preserved in the moment of the message delivery.

Reliability is achieved by application of the transport protocol and, in cases of node crashes, by the System State Agreement procedure. The new module implements the scheme of consistency maintenance described in the chapter 8. It is assured that messages sent by the module are delivered reliably according to the specified delivery requirements. Unrecoverable failures are detected. An up-to-date view of the presence of nodes in a system is maintained. Nodes are allowed to join and leave a system on-the-fly. Node crashes are detected. In cases when a node is erroneously detected as dead, a system is able to recover to a consistent state.

9.1.4 The Module's Structure Overview

The communication module consists of three parts (Figure 9.1). The first part (**BROAD**) is a communication subsystem: it implements the broadcast protocols and enables broadcast communication in the system. The second part (**SYSTEM**) is a system consistency maintenance subsystem. It implements the System State Agreement procedure and enables detection and recovery from node crashes. The third part of the module is **FILTER** which assures that messages delivered ti a node may not cause inconsistencies. The second task of FILTER is detecting unknown nodes and nodes in state inconsistent with the local state. In the figure 9.1, all parts of the IPC module are flow of information among them are depicted.

The IPC module has three interfaces corresponding to its two parts: one with a network and one with other parts of a kernel. The network interface, implemented by the FILTER and BROAD modules, permits receiving and sending messages. The other parts

The second goal of this project is preparation to implementing a more general interprocess communication module for Emerald. The general module will support broadcast and Wide Area Network communication. The protocols developed in this project are the first step towards the general module.

The third goal is to create a basis for experiments with broadcast communication. Functionality of the new protocols is larger then it is necessary for the language. The extra functionality will be used in experiments aimed at verifying properties of different approaches to network interprocess communication. Results of experiments will be applied in development of the enhanced communication module.

9.1.2 Implementation's Properties

The implementation of the protocols and the communication module should exhibit the following properties:

flexibility — applications differ in their demands, for example, for a necessary level of reliability or asynchrony.