Distributed Peer-to-Peer Control for Harness

A Dissertation
submitted in partial fulfilment of
the requirements for the degree of

MASTER OF SCIENCE

in Parallel and Scientific Computation

in the

Faculty of Science

The University of Reading

by

Christian Engelmann

24. January 2001

Supervisors:

G. A. Geist
Oak Ridge National Laboratory

Prof. U. Metzler
Fachhochschule für Technik
und Wirtschaft Berlin

Acknowledgements

This thesis is based on my work on the distributed peer-to-peer control for the Harness project. The Harness project is a collaborative effort between the Oak Ridge National Laboratory, the University of Tennessee, and the Emory University. I would like to thank Al Geist for inviting me to work on the thesis subject at the Oak Ridge National Laboratory, for the financial support and for suggestions during the work. Special thanks to Steven Scott for his support and suggestions to the distributed system architecture and the state-replication algorithm. Thanks to Jim Kohl for the organizational assistance to get paid every month and to Jens Schwidder, Torsten Wilde and Sreekanth Pannala for their private support. And last but not least, thanks to Vassil Alexandrov at the University of Reading and Uwe Metzler at the FHTW Berlin for their help.

Abstract

Parallel processing, the method of cutting down a large computational problem into many small tasks which are solved in parallel, is a field of increasing importance in science. Cost-effective, flexible and efficient simulations of mathematical models of physical, chemical or biological real-world problems are replacing the traditional experimental research. Current software solutions for parallel and scientific computation, like Parallel Virtual Machine and Message Passing Interface, have limitations in handling faults and failures, in utilizing heterogeneous and dynamically changing communication structures, and in enabling migrating or cooperative applications. The current research in heterogeneous adaptable reconfigurable networked systems (Harness) aims to produce the next generation of software solutions for distributed computing.

A high-available and light-weighted distributed virtual machine service provides an encapsulation of a few hundred to a few thousand physical machines in a virtual heterogeneous large scale cluster. A high availability of a service in distributed systems can be achieved by replication of the service state on multiple server processes. If one ore more server processes fails, the surviving ones continue to provide the service because they know the state. Since every member of a distributed virtual machine is part of the distributed virtual machine service state and is able to change this state, a distributed control is needed to replicate the state and maintain its consistency. This distributed control manages state changes as well as the state-replication and the detection of and recovery from faults and failures of server processes.

This work analyzes system architectures currently used in heterogeneous distributed computing by defining terms, conditions and assumptions. It shows that such systems are asynchronous and may use partially synchronous communication to detect and to distinguish different classes of faults and failures. It describes how a high availability of a large scale distributed service on a huge number of servers residing on different geographical locations can be realized. Asynchronous group communication services, such as Reliable Broadcast, Atomic Broadcast, Distributed Agreement and Membership, are analyzed to develop linear scalable algorithms in an unidirectional and in a bidirectional connected asynchronous peer-to-peer ring architecture. A Transaction Control group communication service is introduced as state-replication service.

The system analysis distinguishes different types of distributed systems, where active transactions execute state changes using non-replicated data of one or more servers and inactive transactions report state changes using replicated data only. It is applicable for passive fault-tolerant distributed databases as well as for active fault-tolerant distributed control mechanisms. No control token is used and time stamps are avoided, so that all members of a server group have equal responsibilities and are independent from the system time. A prototype which implements the most complicated Transaction Control algorithm is realized due to the complexity of the distributed system and the early development stage of the introduced algorithms. The prototype is used to obtain practical experience with the state-replication algorithm.

Contents

1 Introduction

1.1 Overview

Parallel processing, the method of cutting down a large computational problem into many small tasks which are solved in parallel, is a field of increasing importance in science. Cost-effective, flexible and efficient simulations of mathematical models of physical, chemical or biological real-world problems, like in fusion energy research, material properties research or human gnome research, using parallel processing are replacing the traditional experimental research. Massively parallel processing homogeneously combines a few hundred to a few thousand CPUs in a single large cabinet connected to hundreds gigabytes of memory. In contrast, distributed computing connects single workstations to heterogeneous clusters, which becomes more and more attractive due to the increasing performance of general-purpose hardware solutions.

Current software solutions, like Parallel Virtual Machine (PVM) [1] and Message Passing Interface (MPI) [2], provide different utilitarian models for personally managing computing resources. The application programming interfaces offer a wide variety of functions to allow messaging, task control, event notification and event handlers. PVM uses virtual machine services separately existing on each processor communicating to each other while MPI is based on direct inter-application communication. Both software solutions have different strengths and weaknesses [3] depending on the system architecture. The virtual machine abstraction of PVM is more suitable for heterogeneous systems (distributed computing) and the faster messaging capability of MPI is more qualified for homogeneous systems (massively parallel processing).

However, both models have general limitations [3, 4] in handling faults and failures, in utilizing heterogeneous and dynamically changing communication structures, and in enabling migrating or cooperative applications. The execution times of computational tasks in high performance computing are getting longer than the Mean Time To Failure (MTTF) of hardware and software components. Fault-tolerance with detection of and recovery from faults and failures for a high availability of services is hard to implement with PVM and MPI. Furthermore, both solutions do not provide a general mechanism for a dynamical adaption to application needs e.g. to the different phases of a simulation: data input, problem setup, calculation, analysis and visualization. An application is forced to fit in a fixed PVM or MPI environment.

The current research in heterogeneous adaptable reconfigurable networked systems (Harness) [5, 6, 7] aims to produce the next generation of software solutions for distributed computing. A high-available and light-weighted distributed virtual machine (DVM) provides an encapsulation of physical machines with a more abstract view. This view is identical to all members of the DVM and contains the current DVM state including the list of computing resources and their availability. Hardware resources can be managed using a distributed control [6, 9, 10] which also provides detection of and recovery from faults and failures. Software resources can be managed using a plug-in mechanism [6, 8, 9] where the DVM builds a software backbone with a public pluggable interface. Several DVMs can cooperate, merge and split dynamically.

A high availability of a service in distributed systems can be achieved by replication of the service state on multiple server processes [11, 12]. If one ore more server processes fails, the surviving ones continue to provide the service because they know the state. The Harness DVM is conceived as such a

high-available service. It will survive until at least one server process is still alive. Since every member of a DVM is automatically a part of the DVM state and is able to change the DVM state, a distributed control is needed to replicate the DVM state and maintain its consistency. This distributed control manages DVM state changes as well as the DVM state-replication and the detection of and recovery from faults and failures of server processes. It is basically a fault-tolerant distributed state-database system, where every server process (member) as a complete and consistent replicated copy of the state-database and is able to process transactions to change it.

1.2 Objectives

This work aims to develop a distributed control for the Harness distributed virtual machine that avoids single point (or set of points) of failure for a distributed heterogeneous system architecture. The design should allow a trade-off between performance and robustness i.e. tunable multi-point failure conditions. The distributed control needs to automatically detect and recover from faults and failures and should be able to detect and recover from faults and failures that occur during a recovery from previous ones (cascaded faults and failures). The main goal is the development, implementation and test of a distributed state-replication algorithm for a consistent fault-tolerant distributed control in a distributed heterogeneous system architecture which meets the criteria of correctness, fault-tolerance, scalability, heterogeneity and efficiency.

The preliminary system design of this work outlines the general distributed system architecture and its behavior by defining terms, conditions and assumptions and analyzing the system requirements. It gives an overview of the distributed system model and the distributed algorithms needed by the distributed control. The implementation strategy answers implementation and integration questions and describes testing techniques for the implementation of a prototype which realizes the distributed state-replication algorithm. The detailed software design gives a detailed description of the implementation and the test results of the previously defined testing techniques. At the end, the success of this work is evaluated against the outlined objectives and the advancement of the state of the art in the related area of fault-tolerant distributed computing.

1.3 Previous Work

Previous research in fault-tolerant distributed systems covers a wide variety of application areas from distributed real-time systems to high-available distributed databases. It differentiates between synchronous and asynchronous fault-tolerant distributed systems by analyzing its communication behavior. Group communication services are introduced to simplify and ensure server group related information exchange despite random communication delays, failures and recoveries [11, 12]. Such group communication services are extended to detect and recover different classes of faults and failures [13, 14, 15, 16, 18, 19, 20, 21, 22] and they are adapted to different kinds of network architectures [17]. The state-replication is divided into active and passive replication, where a trade-off between performance and robustness achieves scalability [23, 24, 25, 26].

This work is mostly based on the previous research on synchronous and asynchronous group communication services [11, 12] of Flaviu Cristian at the Department of Computer Science and Engineering of the University of California in San Diego and the previous research on a distributed control algorithm for the Harness DVM [6, 9, 10] of Steven L. Scott at the Computer Science and Mathematics Division of the Oak Ridge National Laboratory. Other previous work which provides already solutions of state-replication algorithms in distributed systems is evaluated against the requirements of correctness, fault-tolerance, scalability, heterogeneity, efficiency and feasibility in heterogeneous distributed computing. The results of this evaluation process are used in this work to form the distributed system model and the distributed algorithms.

2 Preliminary System Design

2.1 High-Available Distributed System

A system which is designed, implemented and deployed with sufficient components (hardware, software and procedures) to satisfy the system's functional requirements, but no more, has basic availability. A system which is designed, implemented and deployed with sufficient components to satisfy the system's functional requirements but which also has sufficient redundancy in components (hardware, software and procedures) to mask certain defined faults, has high availability [27]. A high-available distributed system provides redundancy of hardware and software components using a high-available server group. If one ore more server fails, the surviving ones sill satisfy the system's functional requirements. The redundancy of software components can be achieved using hot-standby (active replication) or warm-standby (passive replication).

2.1.1 High-Available Server Group

In distributed systems, a server is a processing unit with one or more processors, memory and disk space running an operating system e.g. a single or double processor workstation or a massively parallel processor system. Such a server is connected to a network and provides services to applications running on the same or other servers. A group of servers such as a cluster of workstations or collaborating high-performance computing facilities provides services to parallel applications running on all servers or a subset of the server group. Every server has a local state produced by the states of the available services on the server. The global state of the server group consists of the states of all servers in the group and contains the states of all in the server group available services. A high-available server group provides a high availability of services.

2.1.2 High-Available Service

A hot-standby high availability of a service in distributed systems can be achieved by consistent replication of the service state from one server to all servers in the group. If one ore more server fails, the surviving ones continue to provide the service because they know its current state [11, 12]. A warm-standby high availability of a service in distributed systems can be achieved by keeping a non-permanently maintained backup of the service state of a server at all servers in the group. If one ore more server fails, the surviving ones roll back and restart the service because they know a backup of its state [6, 9, 10]. Group communication services simplify group related information exchange for the maintenance of consistent replication of service states of a server group despite random communication delays, failures and recoveries.

2.1.3 High-Available Distributed State-Database

The global state of a high-available server group is managed with a distributed state-database containing the states of all servers in the group. A state change of a service state is a state change of the local server state and a state change of the local server state is a state change of the global server group state. A state change of the global server group state is always a transaction which modifies

data, stores new data or deletes data depending on the state-database content during execution. Transactions may add or remove a server to or from the server group, start or shutdown a service at all servers or at a subset of the server group, change states of services or report state changes of services. The state-database is completely replicated to provide high availability, but the success of a transaction execution may depend on non-replicated data.

2.1.4 High-Available Distributed Virtual Machine

A single virtual machine service provides an abstract interface for accessing basic server tasks of a single server such as task control, event notification and event handlers. A parallel virtual machine service provides an abstract interface for multiple virtual machine services each running on a single server for accessing collaborative server tasks like messaging and distributed task control. The distributed virtual machine service however provides an abstract interface for accessing basic server tasks and collaborative server tasks with one distributed service running on each server. The number of members (servers) of a virtual machine is one in a single or parallel virtual machine service and can be more than one in a distributed virtual machine service. A high-available distributed virtual machine can be achieved by providing a high availability of its service.

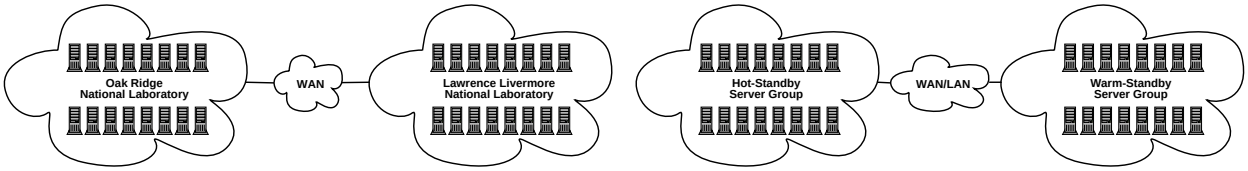
2.2 High Availability for Harness

The Harness distributed virtual machine is conceived as a high-available distributed virtual machine with a high availability of its service. The service provides access to basic server tasks and collaborative server tasks as well, where the light-weighted distributed virtual machine service itself provides only basic functionalities like add or remove a server to or from the server group, start or shutdown a service at all servers or at a subset of the server group, change states of services or report state changes of services. Such services are managed via a plug-in mechanism supported by the distributed virtual machine service. Parallel applications use high-available plug-in services and the high-available distributed virtual machine service to perform computations. The high availability of plug-in services is guaranteed by the high-available distributed virtual machine service.

2.2.1 Distributed System Architecture

Heterogeneous distributed computing combines a few hundred to a few thousand single- or multi-processor workstations to a heterogeneous cluster of workstations. Furthermore, such workstation clusters and massively parallel processing systems can be connected to a heterogeneous large scale super-cluster in collaborative computing, where high-performance computing facilities of different research laboratories at different locations in the world are merged together for a common task. A linear scalable and efficient distributed virtual machine service maintains a hot-standby high availability only at a subset of the server group and a warm-standby high availability at all other servers, so that the server group of a distributed virtual machine always consists of a hot-standby server group and a warm-standby server group [6, 9, 10].

This logical distributed system architecture however requires a part of the hot-standby server group at each cluster or high-performance computing facility i.e. geographical location to provide an equal quality of high-available services to all members of the distributed virtual machine. Every geographical location is able to adjust its own high availability level by choosing the number of servers in the local hot-standby server group depending on the local distributed system capabilities i.e. network and computing performance. So that the more detailed distributed system architecture of the Harness distributed virtual machine consists of local hot-standby server groups and local warm-standby server groups which are connected together and a globally connection of the hot-standby server groups of all geographical locations.



(a) Physical (b) Logical

Figure 2.1: Physical and Logical Distributed System Architecture

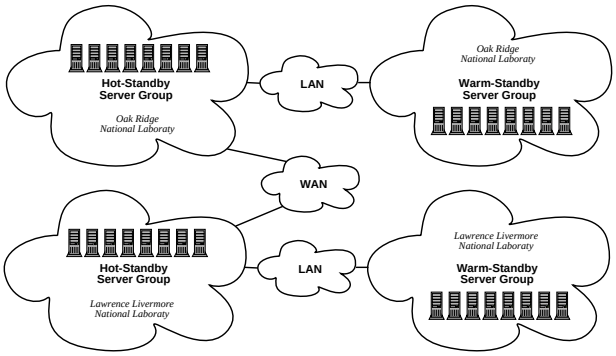


Figure 2.2: More Detailed Distributed System Architecture

2.2.2 Network Architecture

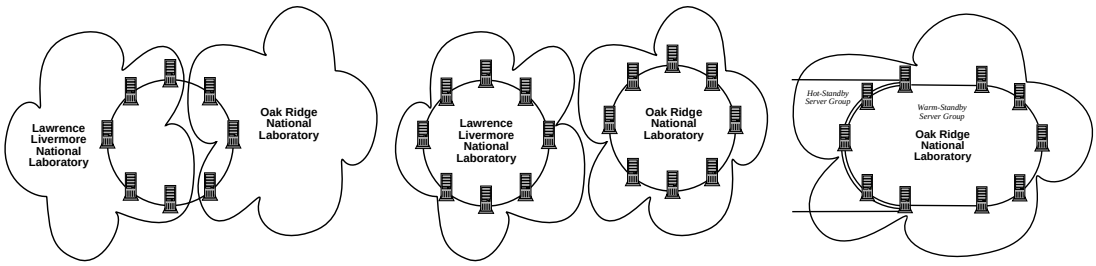
Asynchronous systems make no timing assumptions for communication. There is no bound on message delay, clock drift, or the time necessary to execute a step. These systems are characterized by uncertainty communication i.e. there is no distinction between server crash, server overload, network overload or network breakdown. In contrast, synchronous systems make timing assumptions for communication, there are such bounds to execute a step and these systems are characterized by certainty communication i.e. there is a distinction between different classes of faults and failures. A mixture of both is often used to solve timed real-world problems and to provide real-time properties as well as non real-time properties. Furthermore, fault-tolerant systems use the distinction between different classes of faults and failures for detection and recovery [11, 12].

A heterogeneous distributed system for solving mathematic and scientific problems using parallel processing has no real-time requirements at all, so it is generally asynchronous. But partially synchronous communication can be used to detect communication failures and server faults, which is necessary to provide fault-tolerance. The TCP/IP network protocol provides connection oriented and failure detecting communication. Additionally, the use of TCP/IP satisfies the heterogeneity required by a heterogeneous distributed system due to the wide spread of TCP/IP networks. A server overload, server crash, network overload or network breakdown causes a detection of a TCP/IP communication error which is automatically handled as a temporary server fault. It is recognized as a permanent server fault if the server remains unreachable.

The Harness distributed virtual machine is conceived as a linear scalable distributed virtual machine service i.e. all system resources are linear dependent on the number of members of the distributed virtual machine (number of servers in the server group). This requires all algorithms and the system architecture itself to be linear scalable. Since the network as part of the system architecture is also a system resource, broad- and multicast communication is generally not linear scalable, and singlecast communication is only linear scalable in a linear scalable peer-to-peer network architecture. Most

peer-to-peer network architectures are not linear scalable, because the number of peer-to-peer links per node depends on the number of nodes, like in fully connected networks or hypercubes. Other networks provide linear scalability but complicated structures like meshes and k-nary trees.

Only chain and ring networks are linear scalable with a simple structure, where the ring with a diameter of $D(n) = n/2$ is more efficient than the chain with a diameter of $D(n) = n - 1$. The warm-standby and the hot-standby server groups of the Harness distributed virtual machine are organized in ring architectures. To provide a connection between local warm-standby and hot-standby server groups, every hot-standby server is always part of the warm-standby server group and provides a bridge between both groups. All hot-standby servers independently update the in the warm-standby server group non-permanently maintained backup of the distributed virtual machine state-database via the warm-standby server ring. All warm-standby servers can send transaction requests to the hot-standby server group via the warm-standby server ring.



(a) Hot-Standby (b) Warm-Standby (c) Local Connection

Figure 2.3: Detailed Hot- and Warm-Standby Network Architectures

2.2.3 Distributed Control

The distributed control of the Harness distributed virtual machine service manages the state-replication for the hot- and warm-standby server groups, so that each member of the distributed virtual machine service runs the same distributed control algorithm in hot- and/or warm-standby mode. A server runs the hot-standby mode if it is the first member of a distributed virtual machine or if it was elected by other hot-standby servers e.g. to balance the high availability level after a new member was added to or removed from the distributed virtual machine or after a hot-standby server fault. A server runs the warm-standby mode if it was added to the distributed virtual machine or forced by other hot-standby servers to balance the high availability level. Group communication services are used to maintain the consistency of the state-database in both server groups.

The hot- and warm-standby server modes use similar algorithms which differ only in the update frequency of state-changes and in the dependency of state changes on non-replicated data. The hot-standby mode consistently updates every state-change of the complete server group at all servers in the hot-standby server group to maintain a consistent up-to-date replication. State changes depend on non-replicated data if the state of the complete server group is actively changed. The warm-standby mode consistently updates a non-permanently maintained backup of the state of the complete server group at all servers in the warm-standby server group to maintain a consistent backup replication of the server group. These state changes do not depend on non-replicated data, since the state of the server group is not actively changed.

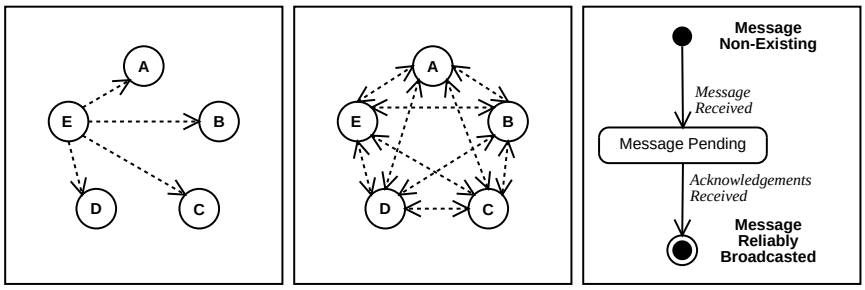
2.3 Group Communication

Group communication services such as Reliable Broadcast, Atomic Broadcast, Distributed Agreement, Transaction Control and Membership simplify maintenance of consistently replicated data in a server group despite random communication delays, failures and recoveries. The Reliable Broadcast provides a service to reliably broadcast messages to all members while the Atomic Broadcast additionally achieves a globally unique order of reliably broadcasted messages. The Distributed Agreement provides an agreement on a decision taken at all members by all members. The Transaction Control combines Atomic Broadcast and Distributed Agreement to provide a linear history of state changes at all members. The Membership manages the server group with member admissions and member removals, where an agreement on an initial state of a server group is achieved.

2.3.1 Reliable Broadcast

The Reliable Broadcast is the most basic group communication service. It ensures a message delivery from one member to all members in a server group, so that each member knows that every member knows the message. The Reliable Broadcast does not ensure that each member knows that every member knows that a message is reliably broadcasted. In order to reliably broadcast a message, one member broadcasts the message to all members including to itself in phase one and each member which receives a message broadcasts an acknowledgement related to the received message back to all members including to itself in phase two. A member knows that a message is reliably broadcasted if the message and the related acknowledgements of all members are received. The Reliable Broadcast performs two phases of broadcast communication.

Since messages and acknowledgements are broadcasted over a network from one member to all members in a server group, receiving order, operation count and communication costs highly depend on the network architecture and the network protocol of the server group. Acknowledgements may be received before messages and messages may be reliably broadcasted asynchronously i.e. at different times. A Reliable Broadcast in a fully connected network architecture with a broadcast network protocol performs $O(n) = 2$ operations with $C(n) = n^2$ communications, where the maximum number of simultaneous communications is n. But a Reliable Broadcast in a bus network architecture without a broadcast network protocol performs $O(n) = n^2$ operations with $C(n) = n^2$ communications, where the maximum number of simultaneous communications is 1.



(a) Phase 1 (b) Phase 2 (c) State-Machine

Figure 2.4: Reliable Broadcast ($O(n) = 2$, $C(n) = n^2$)

The communication system detects communication failures and recovers from them if possible or report member faults otherwise. Transmissions (messages and acknowledgements) can always get lost with a member fault. A correct and complete recovery requires to send all lost transmissions again if they interfere correct members, so that faulty members are excluded from the server group and

messages are reliably broadcasted to all members of the server group despite member faults if at least one correct member knows the message. A correct and complete recovery is based on the local data at every member stored by the Reliable Broadcast. Every member knows which messages are locally pending and which to the pending messages related acknowledgements were received and sent from and to every member.

Previously sent transmissions are sent again during a recovery. Every member knows if a message was locally received before and if an acknowledgement related to a pending message was locally received before. But system resources such as memory and disk space are finite and a member cannot record data about all reliably broadcasted messages. So that a member has generally no information about sent or received acknowledgements related to reliably broadcasted messages, because all related data is deleted. A correct and complete recovery needs to gather the data from all members to sent such previously sent acknowledgements again. A correct but incomplete recovery algorithm leaves messages as not reliably broadcasted by not sending such acknowledgements again to avoid the communication overhead.

2.3.2 Atomic Broadcast

The Atomic Broadcast group communication service is an extension of the Reliable Broadcast. It ensures a globally unique order of broadcasted messages at all members in a server group, so that each member knows that every member knows the message in an unique order i.e. which message precedes another at all members. The Atomic Broadcast ensures also a semantically correct order of broadcasted messages i.e. a message which is caused by a previous message cannot be wrongly ordered before the previous message. In order to atomically broadcast a message, the Reliable Broadcast is extended with a distributed message numbering algorithm which gives every message a globally unique id and a distributed message sorting technique which sorts messages into the by the message numbering algorithm given order.

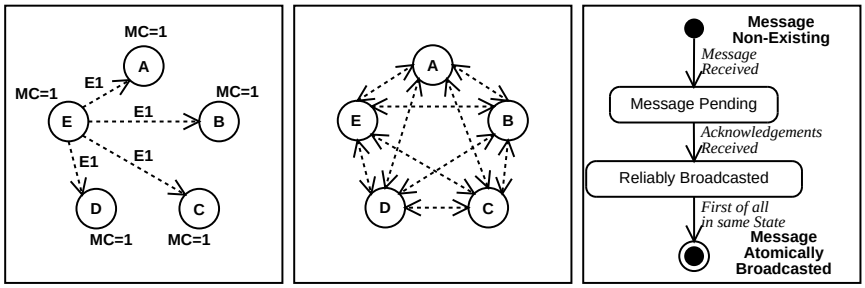
The distributed message numbering algorithm in distributed systems with a global maintained time uses a time stamp and a globally unique member priority to combine them to a globally unique message id. Such a message id contains the time stamp of the injection time and the priority of the member which injected the Atomic Broadcast. Messages are ordered using the time stamp, and messages with equal time stamps are ordered using the member priority. The Atomic Broadcast using timestamps requires a maximum clock drift less than the minimal execution time of an Atomic Broadcast in the distributed system. Otherwise the semantically correct order of messages cannot be maintained. Additionally, it requires that every member has the same global time which is maintained by a separate service.

The distributed message numbering algorithm in distributed systems without a global maintained time uses a global message number which is maintained with the message broadcast and a globally unique member priority, so that a specific message number cannot be reused by a member if it is known. Every member has a local copy of the global message number which is increased before a new Atomic Broadcast is injected. The message id contains the increased message number and the priority of the member which injected the Atomic Broadcast. The local copy of the global message number is updated with higher received values during phase one of the Atomic Broadcast. Messages are ordered using the message number, and messages with equal message numbers are ordered using the member priority.

Two different distributed message sorting techniques can be used to sort messages into the by the message numbering algorithm given order. A communication blocking technique during the phase one of an Atomic Broadcast blocks the broadcast of messages to ensure that they will be received in the right order at every member. However, this violates the Reliable Broadcast by holding a message in an unreliable broadcasted phase. Since a member fault may occur, the information that a message was blocked or forwarded and lost at a faulty member is lost. A recovery algorithm cannot

differentiate between blocked and lost messages unless it reliably gathers all data from all members, so that a correct and complete recovery is required for this technique. Additionally all members are synchronized and all atomic broadcasts of messages are sequentialized.

Another technique is to reliably broadcast messages unordered and sort them at each member by insertion into a sorted list on receiving, so that a message is atomically broadcasted if all previous messages are reliably broadcasted. One atomically broadcasted message may automatically trigger other already reliably broadcasted messages to be in the atomically broadcasted state, so that blocks of messages may occur atomically broadcasted at a member. This technique requires only a correct recovery if unreliable broadcasted messages are recognized and removed. The operation count, communication costs and fault recovery of an Atomic Broadcast are similar to the Reliable Broadcast. The operation count is extended by the waiting for previous messages to be reliably broadcasted which may be different at every member and with every Atomic Broadcast.



(a) Phase 1 (b) Phase 2 (c) State-Machine

Figure 2.5: Atomic Broadcast ($O(n) = 2$, $C(n) = n^2$)

2.3.3 Distributed Agreement

The Distributed Agreement group communication service is based on a Reliable Broadcast executed by all members in a server group. It ensures that a decision is taken at all members by all members, so that each member knows the decision of every member and is able to evaluate it using the same voting algorithm. In order to perform a Distributed Agreement on a decision, every member executes a Reliable Broadcast of its decision. A decision is taken at a member if the decisions of all members are reliably broadcasted. Common voting algorithms are majority, minority (at least one agrees) or strictly agreement. A Distributed Agreement in a fully connected network with a broadcast network protocol performs $O(n) = n^2$ operations with $C(n) = n^3$ communications, where the maximum number of simultaneous communications is n.

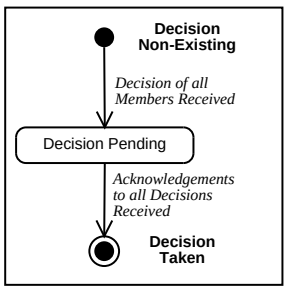


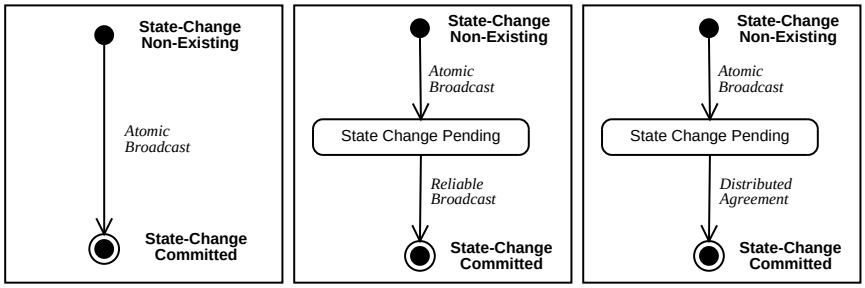
Figure 2.6: Distributed Agreement State-Machine ($O(n) = n^2$, $C(n) = n^3$)

2.3.4 Transaction Control

The Transaction Control group communication service is based on the Atomic Broadcast and may involve additionally the Reliable Broadcast or the Distributed Agreement. It ensures a linear history of state changes, i.e. of executed and committed transactions of a distributed database, at all members of a server group, so that each member knows that every member knows the state changes in an unique and semantically correct order. In order to achieve an agreement on a linear history of state changes at all members, every state change is performed using the Atomic Broadcast of transactions. The Transaction Control additionally involves the Reliable Broadcast or the Distributed Agreement to ensure a correct transaction commit or rejection based on the transaction execution result for active state changes which are based on non-replicated data.

A distributed system which supports inactive state changes which are based only on replicated data executes an Atomic Broadcast of a transaction. A transaction is executed if it is atomically broadcasted and committed if the local transaction execution result is positive. The execution of one state change performs $O(n) = 2$ operations with $C(n) = n^2$ communications in a fully connected network with a broadcast network protocol, where the maximum number of simultaneous communications is n. Such a distributed system executes and commits illegal transactions if they are based on non-replicated data and revert them using state changes, so that the global state is consistently replicated but not always correct. This situation occurs if a member admission or a service start fails and not at all members are able to recognize the failed transaction execution.

A distributed system which supports active state changes which are based on non-replicated data of only one server executes an Atomic Broadcast of a transaction followed by a Reliable Broadcast of the transaction execution result performed by the server with the used non-replicated data. A transaction is executed if it is atomically broadcasted, committed if the transaction execution result is reliably broadcasted and positive, and rejected if the transaction execution result is reliably broadcasted but negative. The execution of one state change performs $O(n) = 4$ operations with $C(n) = 2n^2$ communications in a fully connected network with a broadcast network protocol, where the maximum number of simultaneous communications is n. Illegal transactions are executed, committed and reverted if they are based on non-replicated data of more than one servers.



(a) Simple (b) Followed RB (c) Followed DA

Figure 2.7: Transaction Control State-Machines

A distributed system which supports active and inactive state changes which are based on replicated data or non-replicated data of one or more servers executes an Atomic Broadcast of a transaction followed by a Distributed Agreement on the transaction execution result. A transaction is executed if it is atomically broadcasted, committed if all members agree on the transaction execution result and rejected otherwise. The voting algorithm may depend on the transaction type. The execution of one state change performs $O(n) = 2n^2 + 2$ operations with $C(n) = 2n^3 + n^2$ communications in a fully connected network with a broadcast network protocol, where the maximum number of simultaneous

communications is n. Both algorithms for active state changes reliably broadcast the final transaction execution result to provide consistency.

2.3.5 Membership

The Membership group communication service manages the server group with member admissions and member removals. It ensures an agreement on an initial state and a linear history of state changes at all members, so that each member has a complete and consistent replicated state. In order to achieve an agreement on an initial state at all members, every new member obtains a copy of the state of a group member. In order to achieve an agreement on a linear history of state changes at all members, every state change is performed using the Transaction Control group communication service. State changes consistently replicate also the list of members and network architecture information, which are used by the Membership group communication service to reorganize the network architecture and to request information from correct members.

In order to admit a new member to the server group, the Membership group communication service provides an admit state change. The new member is connected and its state is updated during the state change execution. This requires that the new member expects the admission. The new member is able to inject state changes if the admit state change was committed. The admit state change is not committed or reverted depending on the used Transaction Control group communication service type if the member cannot be connected or fails during state change execution. So that the distributed database may consistently contain the wrong entry that the member was admitted until it is removed by an injected fault state change in a distributed system which does not support active state changes which are based on non-replicated data of one or more servers.

In order to remove an admitted member from the server group, the Membership group communication service provides a removal state change. The member is disconnected during the state change execution and the removal state change is generally committed if the member is part of the server group. The removal state change is generally not committed if the member is not part of the server group. Since the list of members is replicated knowledge, a removal state change is always correct. Faulty members are removed from the group of servers using a fault state change which performs the removal and a member fault notification if a member fault was recognized and recovered. Since a recovery algorithm is correct and complete, a member removal can be executed as a controlled member fault without member fault notification.

Member joins, i.e. member admissions of servers which are already members of an other server group of an other distributed virtual machine, or member separation, i.e. removals of members which continue the same services in a different instance of a distributed virtual machine are part of join or split operations of distributed virtual machines. The Membership group communication service provides only the management of single server groups of one distributed virtual machine. Multiple server group or distributed virtual machine operations are managed by other services on a higher abstraction level using the Transaction Control and Membership group communication services. Collaboration, merging and splitting of distributed virtual machines in Harness is a separate part of the collaborative research effort in the Harness team [5, 6].

2.4 Group Communication in an Unidirectional Ring

Since all group communication services are based on broadcasting information to all members of a server group and may perform $O(n) = 2n^2 + 2$ operations with $C(n) = 2n^3 + n^2$ communications in a fully connected network with a broadcast network protocol, a linear scalable peer-to-peer network architecture is needed to provide linear scalable algorithms for group communication services. The ring is such a linear scalable peer-to-peer network architecture where n communication links exists for n members with n or $2n$ simultaneous communications depending on the link properties. A server ring is asynchronously unidirectional connected if all links provide only half-duplex communication and is asynchronously bidirectional connected if all links provide full-duplex communication, since a half-duplex communication link may be used bidirectional but not asynchronously.

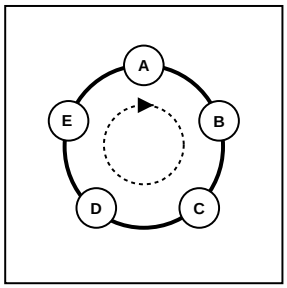


Figure 2.8: Unidirectional Ring Architecture

Nevertheless, an asynchronous member behavior is required to satisfy the asynchronous system property. That requires transmissions to be sent and received asynchronously i.e. without any handshaking. The TCP/IP network protocol provides a sufficient asynchronous communication with failure detection using a partially synchronization by applying the sliding window strategy. The group communication in an unidirectional ring uses constantly established TCP/IP links between members, where only one direction is used. Since TCP/IP links are normally established between a server and a client, every member runs a server connected to the client of the previous member in the ring. The maximum number of transmissions sent by a member but not received by the neighbor is bound on the size of the TCP/IP buffers, where other communication reduces this bound additionally.

2.4.1 Reliable Broadcast

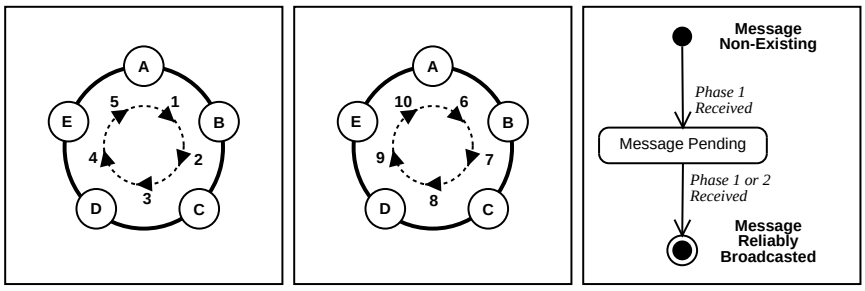
The Reliable Broadcast in an unidirectional ring architecture consists of two phases where transmissions are forwarded from one member to the next. In phase one, a member sends a phase one transmission to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase one to phase two. In phase two, a member sends a phase two transmission related to the phase one transmission to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Reliable Broadcast and each member knows the phase one transmission which represents the message and the phase two transmission which represents the acknowledgement related to the message. So that each member knows that every member knows the message.

Phase one and two transmissions are always related to each other, so that both have the same id but different types. Every member stores an incoming phase one transmission in a list and removes it if the related phase two transmission is received. A phase one transmission is forwarded if it was not in the list before and a phase two transmission is forwarded if the related phase one transmission is in the list. A transition from phase one to two is performed if an incoming phase one transmission was in the list before. The algorithm ends if the related phase one transmission to an incoming phase two transmission is not in the list. Since a phase two transmission exists only if all members know

the related phase one transmission, the first member which knows that a phase one transmission is reliably broadcasted is the member which performs the transition from phase one to two.

Members may reliably broadcast messages simultaneously, so that the Reliable Broadcast is performed simultaneously. But every member is able to execute only a specific amount x of simultaneous Reliable Broadcasts. This bounds the number of simultaneous communications in the ring to to xn and protects members against starvation due to message flooding, which means that a member may be unable to use the Reliable Broadcast group communication service because other members overloaded the system. Since no member is more important than any other, a fair share of the Reliable Broadcast group communication service is provided. So that every member is able to inject a phase one transmission into the ring only after trying to receive a transmission from its neighbor. This interleaves new transmissions with other transmissions in the ring.

The Reliable Broadcast in an unidirectional ring architecture performs $O(n) = 2n$ operations with $C(n) = 2n$ communications, where the maximum number of simultaneous communications is 1. Since $O(n)$ and $C(n)$ are linear dependent on n and the number of simultaneous communications is not dependent on n, this algorithm of a Reliable Broadcast in an unidirectional ring architecture is linear scalable w.r.t. the use of system resources grows linear with the number of members. The simultaneous Reliable Broadcast in an unidirectional ring architecture performs $O(n) = 2n + x$ operations with $C(n) = 2xn^2$ communications, where the maximum number of simultaneous communications is xn. This non-linear communication behavior is similar to the behavior of an unreliable (normal) simultaneous broadcast, where every member simultaneously broadcasts x messages.



(a) Phase 1 (b) Phase 2 (c) State-Machine

Figure 2.9: Reliable Broadcast in an Unidirectional Ring ($O(n) = 2n$, $C(n) = 2n$)

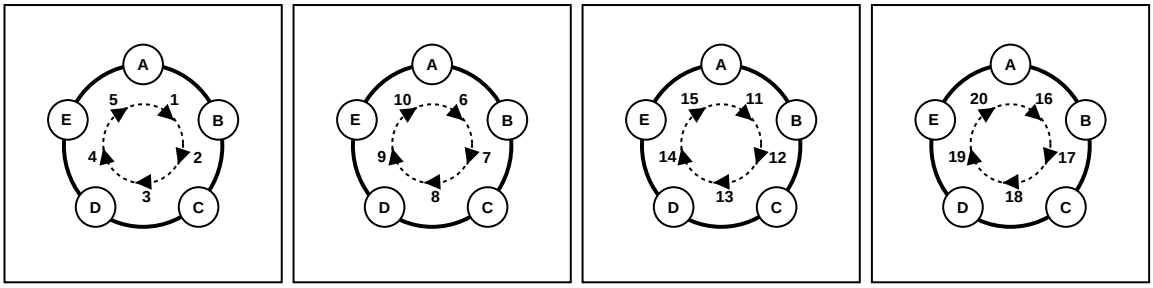
2.4.2 Dependent Reliable Broadcasts

Two ore more Reliable Broadcasts may depend on each other if reliably broadcasted information causes a new information to be reliably broadcasted. Since every member has the same knowledge, the second Reliable Broadcast does not need to wait for the completion of the first Reliable Broadcast in form of two sequentialized Reliable Broadcasts in four phases. These four phases can be interleaved into three phases, so that the phase one of the second Reliable Broadcast is the phase two of the first Reliable Broadcast. The member which performs the transition from phase one to two of the first Reliable Broadcast injects the phase one transmission of the second one. Since every member knows that a second Reliable Broadcast is caused by the first Reliable Broadcast, the phase one transmission of the second Reliable Broadcast is replacing the phase two transmission of the first one.

In such a three-phase Reliable Broadcast, every member stores an incoming phase one or two transmission in a list and removes it if the related phase three transmission is received. So that a phase one transmission is forwarded if it was not in the list before, a phase two transmission

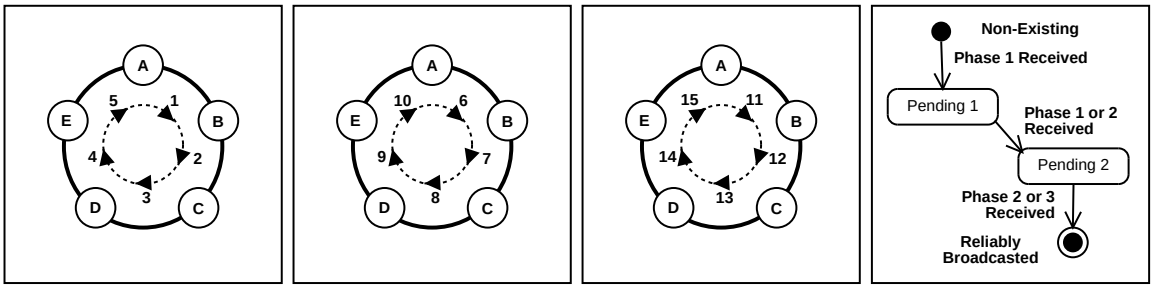
is forwarded if the related phase one transmission is in the list and a phase three transmission is forwarded if the related phase two transmission is in the list. A transition from phase one to two is performed if an incoming phase one transmission was in the list before and a transition from phase two to three is performed if an incoming phase two transmission was in the list before. The algorithm ends if the related phase one transmission to an incoming phase three transmission is not in the list. The phase one and two transmissions are reliably broadcasted similar to the phase one of a two-phase Reliable Broadcast.

Two sequentialized Reliable Broadcasts in an unidirectional ring architecture perform $O(n) = 4n$ operations with $C(n) = 4n$ communications, where the maximum number of simultaneous communications is 1. But two interleaved Reliable Broadcasts in an unidirectional ring architecture perform $O(n) = 3n$ operations with $C(n) = 3n$ communications, where the maximum number of simultaneous communications is 1. Since $O(n)$ and $C(n)$ are linear dependent on n and the number of simultaneous communications is not dependent on n, both algorithms for dependent Reliable Broadcasts in an unidirectional ring architecture are linear scalable. But the interleaved Reliable Broadcast is more efficient than the sequentialized Reliable Broadcast in saving n operation counts and n communications, which is linear to the number of members.



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

Figure 2.10: Sequentialized Reliable Broadcasts in an Unidirectional Ring ($O(n) = 4n$, $C(n) = 4n$)



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) State-Machine

Figure 2.11: Interleaved Reliable Broadcasts in an Unidirectional Ring ($O(n) = 3n$, $C(n) = 3n$)

If every by a member executed Reliable Broadcast causes automatically a second dependent Reliable Broadcast, so that the complete algorithm consists of four phases, these four phases can be interleaved into three phases. If every by a member executed Reliable Broadcast causes automatically a second and a third dependent Reliable Broadcast, so that the complete algorithm consists of six phases, these six phases can be interleaved into four phases. Since Reliable Broadcasts are inter-

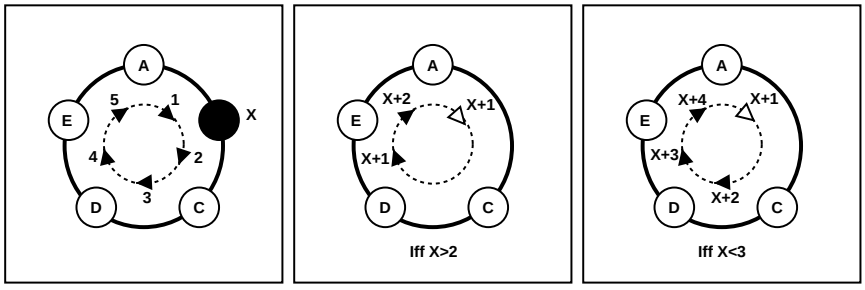
leaved, every phase, except phase one, is a phase two of a two-phase Reliable Broadcast. So that every transmission, except the last-phase, is reliably broadcasted like a phase one in the two-phase Reliable Broadcast. A correct and complete recovery for a two-phase Reliable Broadcast is also a correct and complete recovery for any interleaved p-phase algorithm for dependent Reliable Broadcasts.

2.4.3 Member Fault Recovery

Since member faults may occur, the Reliable Broadcast needs to recover from them in a fault-tolerant distributed system. When a member fault is recognized by the communication system, the two neighbor members of the faulty member connect to each other. The neighbor member which had sent transmissions to the faulty member must assume that all transmissions of the current phase are lost and must be sent again, because previous phase transmissions are always reliably broadcasted. The neighbor member which had received transmissions from the faulty member is able to filter already received transmissions, because it knows which transmissions were received before the member fault. Every transmission which is sent again has additionally a recovery flag, so that phase transitions are performed only once. All transmissions can be sent again except the last-phase.

After sending the last-phase transmission every information about this and the related transmissions are deleted, because system resources are finite and a member cannot record information about all transmissions in its life time. But the neighbor member which had received transmissions from the faulty member knows which last-phase transmissions are pending. This information must be reliably sent to the other neighbor member, so that it can decide which last-phase transmissions are lost. The only way in an unidirectional ring architecture to do so is to send this information in a special recovery transmission over the ring. The lost last-phase transmissions are sent again if the neighbor member receives this recovery message. Another solution is to temporarily reverse the link direction between both members and send the recovery transmission directly.

A recovery of lost phase one transmissions which are not received by any correct member before the fault is impossible, because all information about these transmissions is lost. The member fault recovery without a temporary bidirectional connection between both members performs $O(n) = n + xn + 1$ operations with $C(n) = nx + 1$ communications, where the maximum number of simultaneous communications is 1. The member fault recovery with a temporary bidirectional connection between both members performs $O(n) = nx + 2$ operations with $C(n) = nx + 1$ communications, where the maximum number of simultaneous communications is 1. Both algorithms are linear scalable, but the member fault recovery with a temporary reversed connection between both members is more efficient in saving n operation counts, which is linear to the number of members.

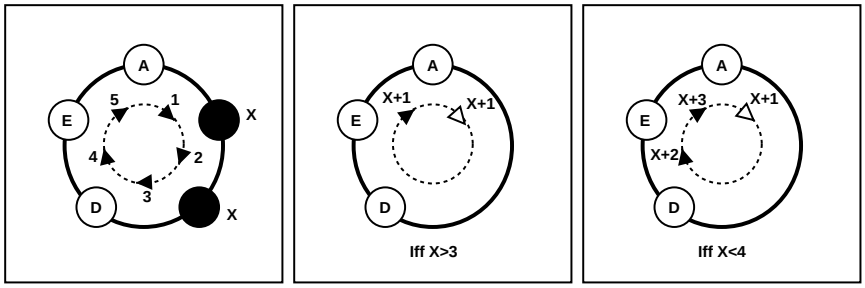


(a) Fault (b) Filtered (c) Recovered

Figure 2.12: Member Fault Recovery in an Unidirectional Ring

2.4.4 Multiple Member Faults Recovery

More than one member may fail at the same time. A group of faulty members which are neighbors build a section of faulty members. Every recovery for a section of faulty members in the ring is managed by the correct neighbors in the same way like a single member fault. The correct members are connected to each other and start the sending of lost transmissions. Since the single member fault recovery assumes that all current-phase transmissions are lost and must be sent again and the lost last-phase transmissions are also sent again, the recovery for a section of faulty members is correct and complete. This recovery applies to every section of faulty members if more than one occurs. Since the ring communication is established immediately after a fault, simultaneous recoveries do not interfere each other despite sending special recovery transmissions.

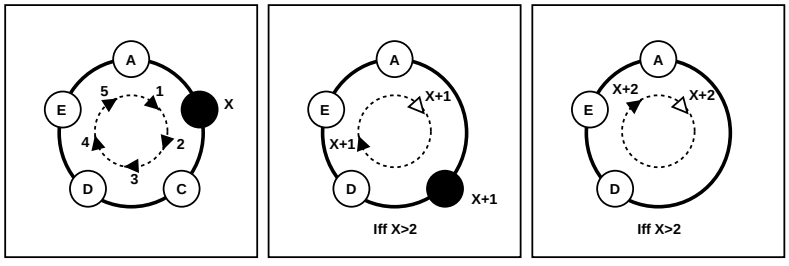


(a) Faults 1 and 2 (b) Filtered (c) Recovered

Figure 2.13: Multiple Member Faults Recovery in an Unidirectional Ring

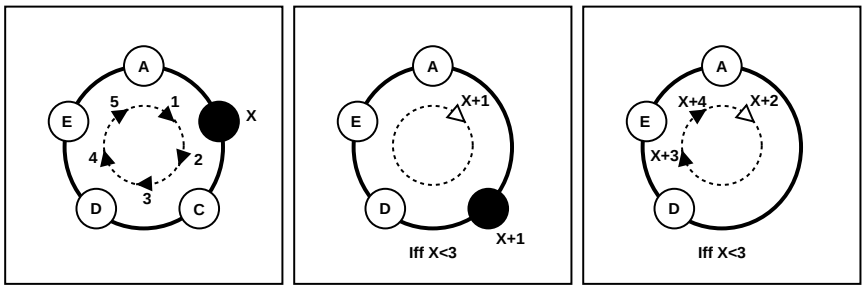
2.4.5 Cascaded Member Faults Recovery

Member faults may also occur during recoveries from them in form of cascaded member faults. Since the single member fault recovery assumes that all current-phase transmissions are lost and have to be sent again, the single fault recovery restarts if one or both neighbor members which perform the recovery fail. All transmissions which are sent during the previous recovery are sent again by the new neighbor member which performs the restarted recovery. The overall operation count and number of communications are similar to (but less than) the operation count and number of communications of a repeated single or multiple member fault recovery for every single or multiple fault occurring during a recovery. But a recovery of lost phase one transmissions which are not received by any correct member before any member fault is still impossible.



(a) Fault 1 (b) Fault 2 (c) Filtered

Figure 2.14: Cascaded Member Faults Recovery in an Unidirectional Ring (1)

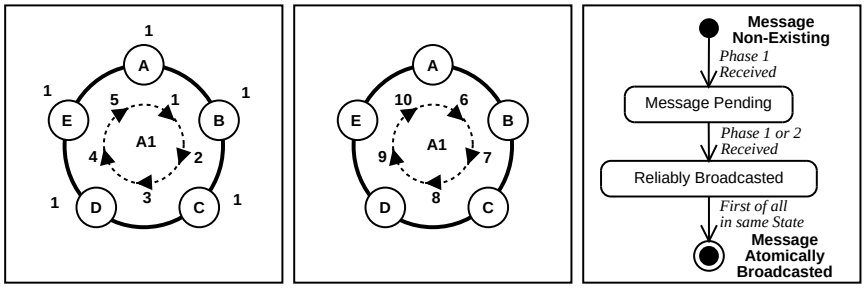


(a) Fault 1 (b) Fault 2 (c) Recovered

Figure 2.15: Cascaded Member Faults Recovery in an Unidirectional Ring (2)

2.4.6 Atomic Broadcast

The Atomic Broadcast in an unidirectional ring architecture extends the Reliable Broadcast in an unidirectional ring architecture by a numbering algorithm and a sorting technique. Every member has a local copy of a global number which is increased if a new Atomic Broadcast is started. The local copy is updated with higher received values. Reliably broadcasted phase one transmissions are ordered by the number, and by the priority on equal numbers. They are sorted at each member by insertion into a sorted list on receiving. So that a reliably broadcasted phase one transmission is atomically broadcasted if all previous phase one transmissions are reliably broadcasted without any blocking. This Atomic Broadcast provides a correct, portable and heterogeneous algorithm since time stamps are not used and phase one transmission blocking is avoided.



(a) Phase 1 (b) Phase 2 (c) State-Machine

Figure 2.16: Atomic Broadcast in an Unidirectional Ring ($O(n) = 2n$, $C(n) = 2n$)

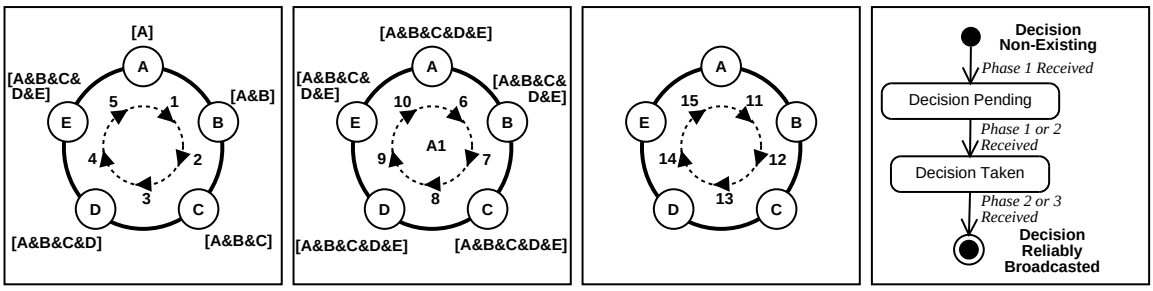
The local copy of the global number is increased and the transmission id with number and member priority is created before a phase one of the Atomic Broadcast starts. In phase one, a member sends a phase one transmission containing the transmission id to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase one to phase two. In phase two, a member sends a phase two transmission containing the transmission id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Atomic Broadcast and each member knows the phase one transmission with transmission number and member priority. So that each member knows that every member knows the phase one transmission and its globally unique order.

A phase one transmission is atomically broadcasted and both transmissions are removed from the list if no phase one transmission with lower number or equal number but lower member priority is pending i.e. not reliably and atomically broadcasted. Operation count, communication costs and fault recovery are similar to the Reliable Broadcast group communication service. The operation count is extended by the waiting for previous phase one transmissions to be reliably broadcasted which may be different at every member and with every Atomic Broadcast.

2.4.7 Distributed Agreement

The Distributed Agreement group communication service in an unidirectional ring architecture is based on two interleaved Reliable Broadcasts in an unidirectional ring architecture. Collective communication is used to avoid n-times Reliable Broadcasts, so that only one Reliable Broadcast is executed which collects the decisions of all members in phase one and distributes the final decision to all members in phase two. Since the Distributed Agreement requires a Reliable Broadcast of the final result, a second Reliable Broadcast of the final result is executed. Two interleaved Reliable Broadcasts are used to save n operation counts. The first Reliable Broadcast realizes also the voting algorithm by combining the decisions using a logical AND for a strictly agreement and a logical OR for minority (at least one) agreement.

The collection of the result for a majority agreement is performed by increasing or decreasing a value, where the final value indicates an agreement if positive, a rejection if negative and and a zero value if undecided. Since the member fault recovery is correct and complete all phases are delivered to all members despite faults. The regeneration of a phase two transmission in a recovery needs to know the final decision. This is possible if every member stores the received final decision locally until a phase three transmission is received. But a special recovery message to know that a phase three transmission went lost is still necessary. So that in phase one, a member sends a phase one transmission containing the combined decision to its neighbor in the ring until it is received by a member which has it already received before.



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) State-Machine

Figure 2.17: Distributed Agreement in an Unidirectional Ring ($O(n) = 3n$, $C(n) = 3n$)

This condition triggers the transition from phase one to phase two. In phase two, a member sends a phase two transmission containing final decision to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase two to three. In phase three, a member sends a phase three transmission containing an acknowledgement to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Distributed Agreement and each member knows that every member knows the final decision. The Distributed Agreement in an unidirectional ring architecture performs $O(n) = 3n$ operations with $C(n) = 3n$ communications, where the maximum number of simultaneous communications is 1.

The transmission size does not depend on the number of members in the ring due to the use of collective communication. So that new members are able to be involved in already running Distributed Agreements, because the algorithm is scalable at any time.

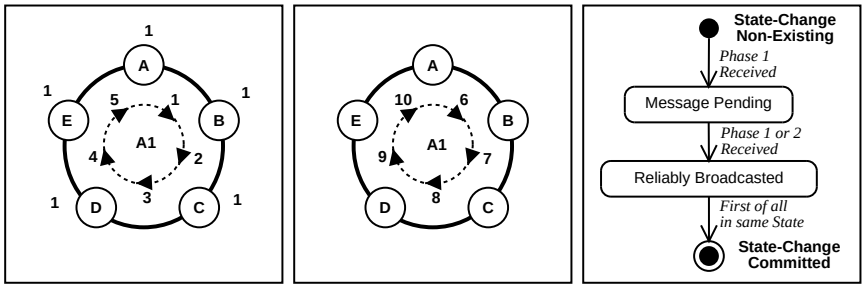
2.4.8 Transaction Control

The Transaction Control group communication service in an unidirectional ring architecture is based on the Atomic Broadcast in an unidirectional ring architecture and may involve additionally the Reliable Broadcast or the Distributed Agreement in an unidirectional ring architecture. The Transaction Control involves the Reliable Broadcast or the Distributed Agreement to ensure a correct transaction commit or rejection based on the transaction execution result for active state changes which are based on non-replicated data, like the member admission state change. The Atomic Broadcast also ensures a globally unique order of a followed Reliable Broadcast or Distributed Agreement, since transmissions cannot pass each other in a ring. The Transaction Control in an unidirectional ring architecture is always linear scalable, since all used group communication services are linear scalable.

Simple Transaction Control

A distributed system which supports inactive state changes which are based only on replicated data executes an Atomic Broadcast of a transaction in an unidirectional ring architecture. A transaction is executed if it is atomically broadcasted and committed if the transaction execution result is positive. The execution of one state change performs $O(n) = 2n$ operations with $C(n) = 2n$ communications, where the maximum number of simultaneous communications is 1. Such a distributed system executes and commits illegal transactions when they are based on non-replicated data and revert them using state changes, so that the global state is consistently replicated but not always correct. In phase one, a member sends a phase one transmission to its neighbor in the ring containing the state change and the transaction id until it is received by a member which has it already received before.

This condition triggers the transition from phase one to phase two. In phase two, a member sends a phase two transmission related to the phase one transmission containing the transaction id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Transaction Control and each member executes and commits or rejects the state change based on the local execution result if it is atomically broadcasted.



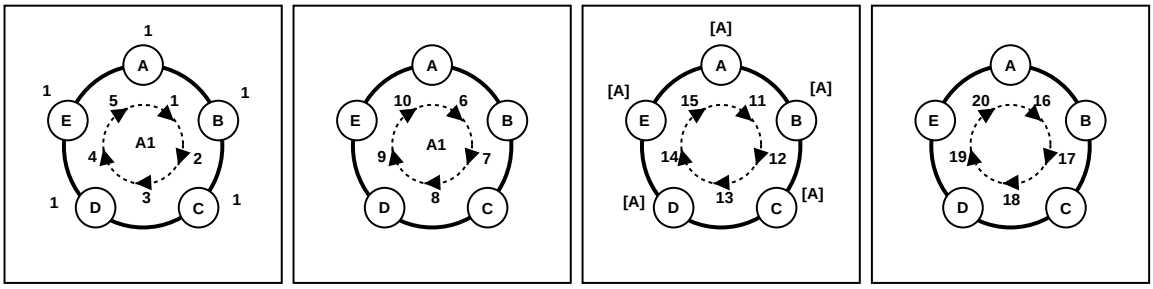
(a) Phase 1 (b) Phase 2 (c) State-Machine

Figure 2.18: Simple Transaction Control in an Unidirectional Ring ($O(n) = 2n$, $C(n) = 2n$)

Transaction Control with Reliable Broadcast

A distributed system which supports active state changes which are based on non-replicated data of only one server executes an Atomic Broadcast of a transaction in an unidirectional ring architecture followed by a Reliable Broadcast in an unidirectional ring architecture of the transaction execution result performed by the server with the used non-replicated data. A transaction is executed if it is atomically broadcasted, committed if the transaction execution result is reliably broadcasted and positive, and rejected if the transaction execution result is reliably broadcasted but negative. Illegal transactions are executed, committed and reverted when they are based on non-replicated data of more than one members. The Atomic Broadcast and the Reliable Broadcast cannot be interleaved, since the member which works with non-replicated data may not always the same.

The execution of one state change performs $O(n) = 4n$ operations with $C(n) = 4n$ communications, where the maximum number of simultaneous communications is 1. In phase one, a member sends a phase one transmission to its neighbor in the ring containing the state change and the transaction id until it is received by a member which has it already received before. This condition triggers the transition from phase one to phase two. In phase two, a member sends a phase two transmission related to the phase one transmission containing the transaction id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Atomic Broadcast and each member executes the state change if it is atomically broadcasted.



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

Figure 2.19: Transaction Control with Reliable Broadcast ($O(n) = 4n$, $C(n) = 4n$)

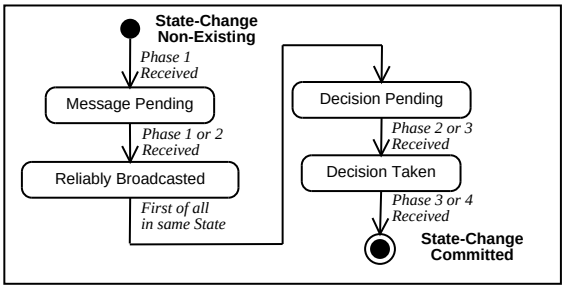


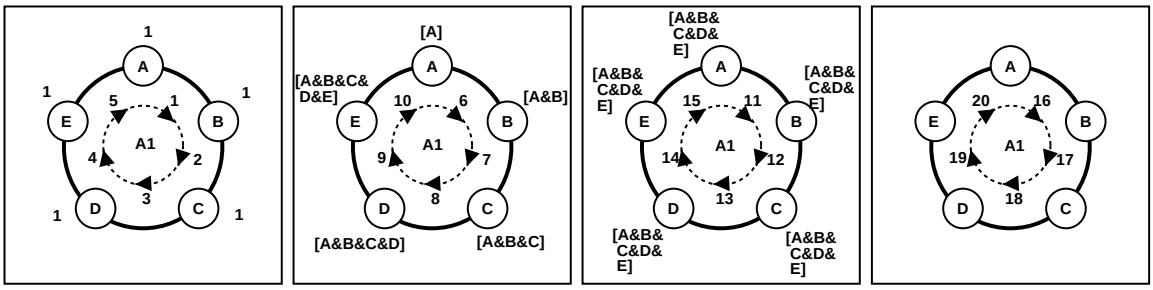
Figure 2.20: State-Machine of Transaction Control with Reliable Broadcast

The Reliable Broadcast is injected by the member which executed the state change based on non-replicated data, where the Reliable Broadcast begins with phase three. In phase three, a member sends a phase three transmission related to the phase two transmission containing the transaction id and the transaction execution result to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase three to phase

four. The state change is committed or rejected based on the in phase three received execution result during phase four. In phase four, a member sends a phase four transmission related to the phase three transmission containing the transaction id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Transaction Control and each member has executed and committed or rejected the state change.

Transaction Control with Distributed Agreement

A distributed system which supports inactive and active state changes based on replicated data or non-replicated data of one or more servers executes an Atomic Broadcast of a transaction followed by a Distributed Agreement on the execution result in an unidirectional ring architecture. A transaction is executed if it is atomically broadcasted, committed if all members agree on the execution result and rejected otherwise. The voting algorithm may depend on the transaction type. The Atomic Broadcast and the Distributed Agreement can always be interleaved if the second phase of the Atomic Broadcast is blocked during the transition from phase one to two until the transaction is atomically broadcasted i.e. all previous transactions are unblocked, so that a transaction is executed and its result is collected during phase two of the interleaved algorithm.



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

Figure 2.21: Transaction Control with Distributed Agreement ($O(n) = 4n$, $C(n) = 4n$)

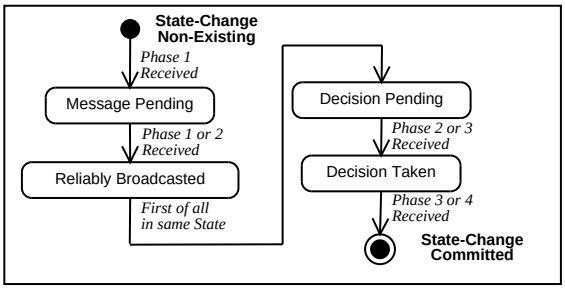


Figure 2.22: State-Machine of Transaction Control with Distributed Agreement

The execution of one state change performs $O(n) = 4n$ operations with $C(n) = 4n$ communications, where the maximum number of simultaneous communications is 1. In phase one, a member sends a phase one transmission to its neighbor in the ring containing the state change and the transaction id until it is received by a member which has it already received before. The transition from phase one to two is performed if and only if no phase one transmission with lower number or equal number but lower priority is pending i.e. not reliably and atomically broadcasted. The phase one transmissions of all other state changes terminate. This is safe since the phase one is reliably broad-

casted and the communication blocking takes place in a second phase of a Reliable Broadcast i.e. all members have this state change in a blocked state and are waiting for the release.

The state change is executed during phase two and its execution result is collected using collective communication. Every member stores locally the result value sent to the next member. In phase two, a member sends a phase two transmission related to the phase one transmission containing the transaction id and the collected transaction execution result to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Atomic Broadcast and each member has executed the state change and the results are collected into the final result. In phase three, a member sends a phase three transmission related to the phase two transmission containing the transaction id and the final transaction execution result to its neighbor in the ring until it is received by a member which has it already received before.

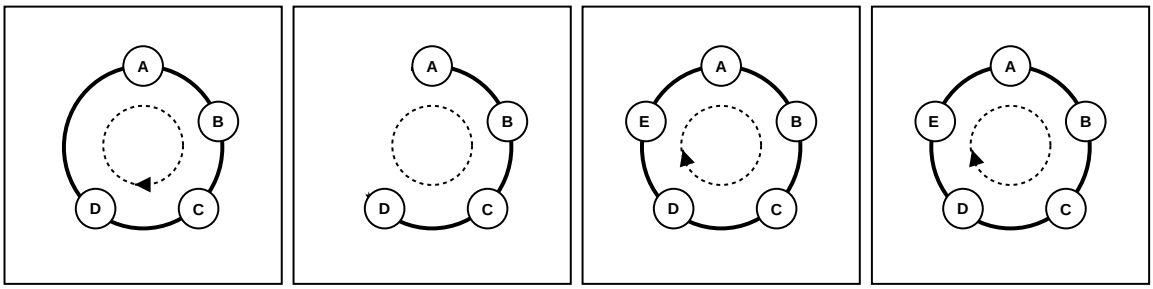
This condition triggers the transition from phase three to phase four. The state change is committed or rejected based on the in phase three received execution result during phase four. In phase four, a member sends a phase four transmission related to the phase three transmission containing the transaction id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Transaction Control and each member has executed and committed or rejected the state change.

Since phase one transmissions appear to be atomically broadcasted in blocks, the transmission phases two, three and four are applied to such a block of atomically broadcasted phase one transmissions. So that a block of transactions is executed in the right order and a block of execution results is collected in phase two, and a block of execution results is reliably broadcasted with the phases three and four, where a block of transactions is committed in the right order in phase four. Such a block consists of simultaneously injected transactions only i.e. transactions with the same number. This block processing technique saves the time of sending x-1 messages for a block of x transactions by sending a message which consists of x results and the id of the leading transaction.

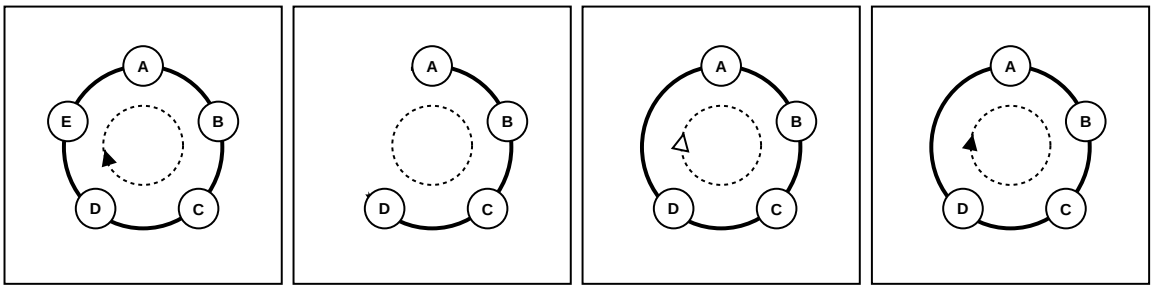
2.4.9 Membership

A member is added to the server group using an 'admit member' state change. During the execution of this state change in an unidirectional ring architecture, the dedicated previous member and the dedicated next member of the new member disconnect each other and try to connect the new member. The previous member sends its complete state to the new member on success and the member is added to the server group. The previous and the next member are reconnected and the member admission state change fails at all members if the connection to the new member cannot be established or the new member fails during the state update. Since a state change is executed and committed in different phases, all members maintain an extra local list of members which represents the current server state which may be uncommitted.

A member of the server group is removed using the controlled member fault during the execution of a 'remove member' state change which exactly defines the member to be removed. A member which recognizes that the next member has to be removed waits for a disconnect from this member. A member which recognizes that it has to be removed, it disconnects the links to the previous and the next member in the ring on state change execution. The next member falls automatically into the member fault recovery algorithm. The previous and the next member are reconnected to each other by the member fault recovery algorithm which sends also lost transmissions again and does not report a member fault, since the disconnect occurred during the 'remove member' state change execution and the member fault is always reported by the previous member.



(a) Receive Phase 2 (b) Disconnect (c) Connect/Update (d) Forward Phase 2



(e) Receive Phase 2 (f) Disconnect (g) Recover (h) Forward Phase 2

Figure 2.23: Member Admission (a-d) and Removal (e-h) in an Unidirectional Ring

2.5 Group Communication in a Bidirectional Ring

The bidirectional connected ring provides the capability to asynchronously send transmissions in both directions over the ring. Since group communication services are based on broadcasting transmissions to all members, the number of operations for a broadcast is decreased from $O(n) = n$ in an unidirectional connected ring to $O(n) \approx n/2$ in an bidirectional connected ring while the number of communications is still $C(n) = n$ and the maximum number of simultaneous communications is increased to 2. So that the bidirectional connected ring provides a more efficient broadcast. The asynchronously bidirectional connected ring requires full-duplex communication between members and an asynchronous member behavior i.e non-blocking communication at every member, so that transmissions from both directions are processed as soon as they arrive and in a fair share.

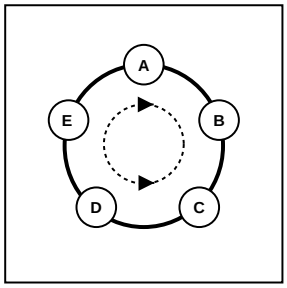
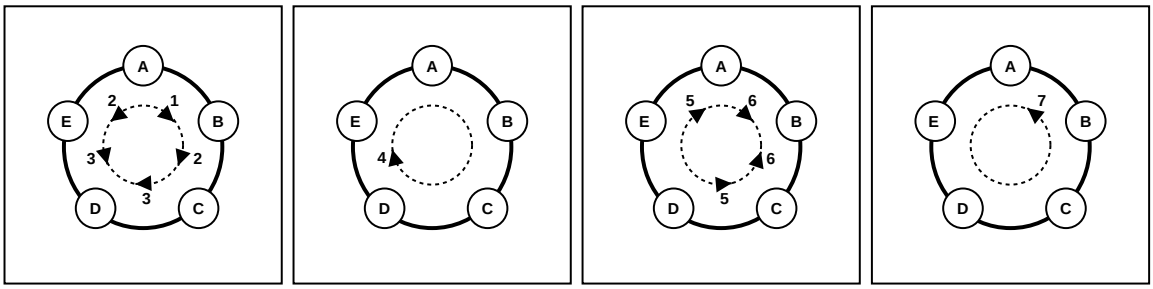


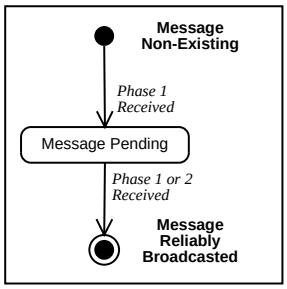
Figure 2.24: Bidirectional Ring Architecture

2.5.1 Reliable Broadcast

The Reliable Broadcast in a bidirectional ring architecture is based on a Reliable Broadcast in an unidirectional ring architecture, but simultaneously executed in both directions i.e. mirrored over the ring to profit from the diameter of the ring of $D(n) = n/2$. Transmissions are forwarded from one member to the next member in both directions over the ring and phase transitions are always performed at two members after transmissions of the same phase and id cross between these two members even if there are only two members in the ring. But one member can only process one transmission at a time, so that a member tries to receive and process transmissions in a fair share from the left member (ring transmissions), from the right member (ring transmissions) and from an external source (transaction requests).



(a) Phase 1 (b) Phase 1 (c) Phase 2 (d) Phase 2



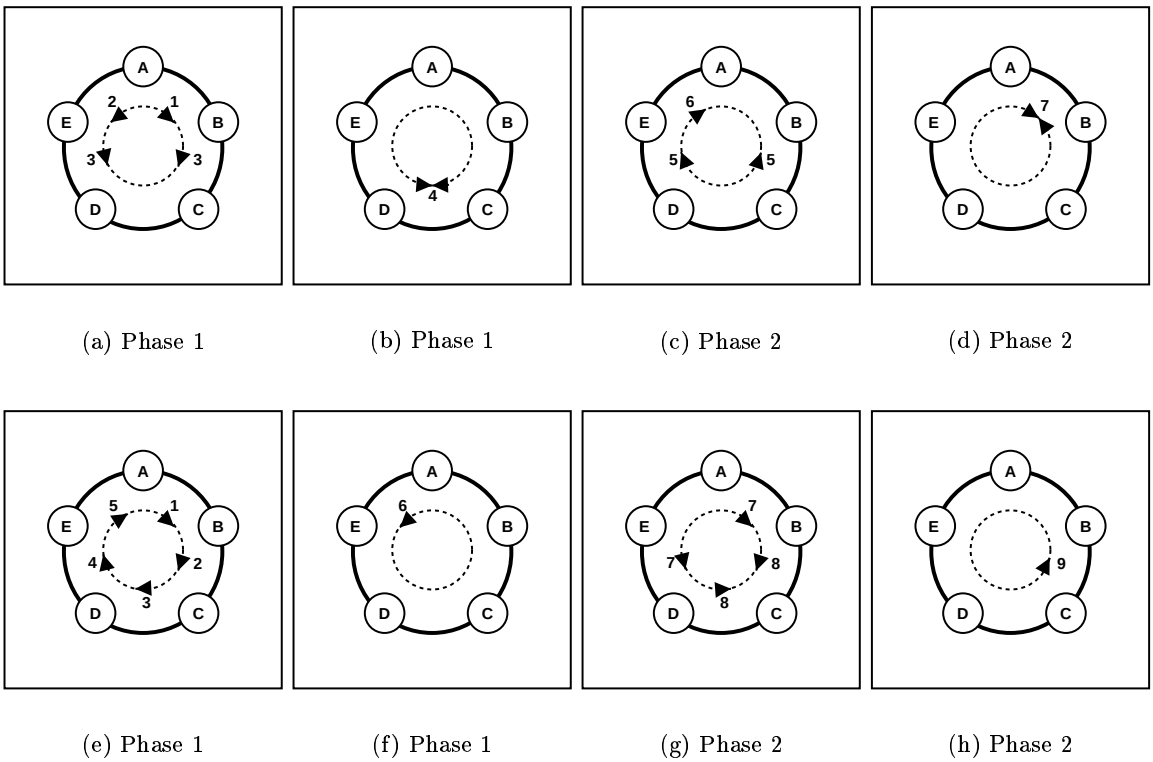
(e) State-Machine

Figure 2.25: Reliable Broadcast in a Bidirectional Ring ($O(n) \approx n$, $C(n) \approx 2n$)

A phase one transmission is injected in both directions by a member. In phase one, a member sends a phase one transmission to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase one to phase two and is always performed at two members. In phase two, a member sends a phase two transmission related to the phase one transmission to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Reliable Broadcast which is always performed at two members. Each member knows the phase one transmission which represents the message and the phase two transmission which represents the acknowledgement related to the message. So that each member knows that every member knows the message.

Similar to the Reliable Broadcast in an unidirectional ring architecture, every member stores an incoming phase one transmission in a list and removes it if the related phase two transmission is received. A phase one transmission is forwarded if it was not in the list before and a phase two transmission is forwarded if the related phase one transmission is in the list. A transition from phase

one to two is performed if an incoming phase one transmission was in the list before. The algorithm ends if the related phase one transmission to an incoming phase two transmission is not in the list. Since a phase two transmission exists only if all members know the related phase one transmission, the first two members which know that a phase one transmission is reliably broadcasted are the members which perform the transition from phase one to two.



(a) Phase 1 (b) Phase 1 (c) Phase 2 (d) Phase 2

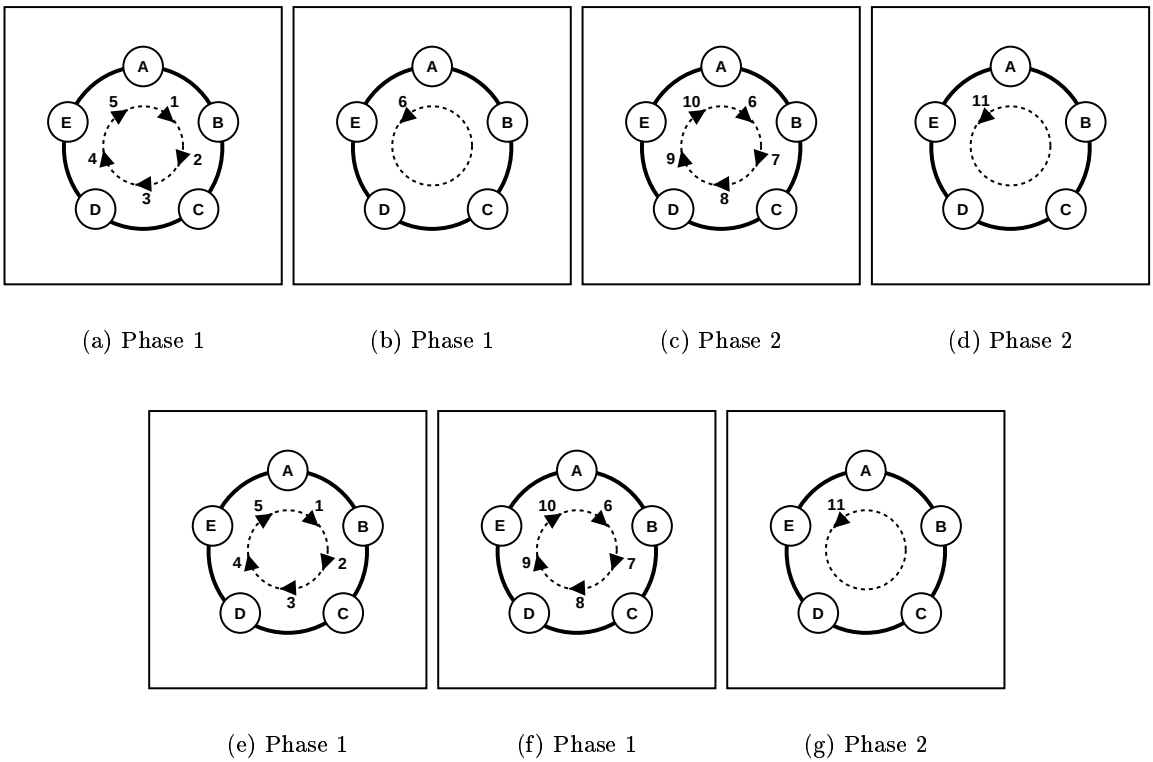
(e) Phase 1 (f) Phase 1 (g) Phase 2 (h) Phase 2

Figure 2.26: Reliable Broadcast in a Bidirectional Ring with Congestion (Balanced)

The Reliable Broadcast in a bidirectional ring architecture performs $O(n) \approx n$ operations with $C(n) \approx 2n$ communications, where the maximum number of simultaneous communications is 2. Since $O(n)$ and $C(n)$ are linear dependent on n and the number of simultaneous communications is not dependent on n, this algorithm of a Reliable Broadcast in a bidirectional ring architecture is linear scalable w.r.t. the use of system resources grows linear with the number of members. The simultaneous Reliable Broadcast in a bidirectional ring architecture performs $O(n) \approx n+x$ operations with $C(n) \approx 2xn^2$ communications, where the maximum number of simultaneous communications is $2xn$. This non-linear communication behavior is similar to the Reliable Broadcast in an unidirectional ring architecture.

Since transmissions are asynchronously forwarded from one member to the next member in both directions over the ring and network congestion may occur, the algorithm for a Reliable Broadcast in a bidirectional ring architecture needs to recognize and correct illegal states produced by too late received transmissions from one direction. As long as the same phase transitions are executed at two members the Reliable Broadcast works correct. This is always guaranteed if a transmission of one direction goes less than two rounds in the ring without meeting the transmission of the other direction i.e. crossing recognized by a member, or w.r.t. if the Reliable Broadcast is not completely executed like in the unidirectional ring architecture. Therefore, a phase one transmission of one direction which was previously reliably broadcasted by the other direction terminates.

Since system resources are finite and a member cannot record information about all transmissions in its life time, this is impossible. So that the Reliable Broadcast in a bidirectional ring architecture is not applicable without an additional algorithm which determines whether a phase one transmission was reliably broadcasted or not. An Atomic Broadcast in a bidirectional ring architecture based on this Reliable Broadcast may implement such an algorithm based on the total order of phase one transmissions. If the total order of phase one transmissions defines a message id which has a one to one relationship to a phase one transmission for all possible phase one transmissions, a previously reliably broadcasted phase one transmission can be determined by comparing its id with the id of the last reliably broadcasted phase one transmission.



(a) Phase 1 (b) Phase 1 (c) Phase 2 (d) Phase 2

(e) Phase 1 (f) Phase 1 (g) Phase 2

Figure 2.27: Reliable Broadcasting a Bidirectional Ring with Congestion (Corrected)

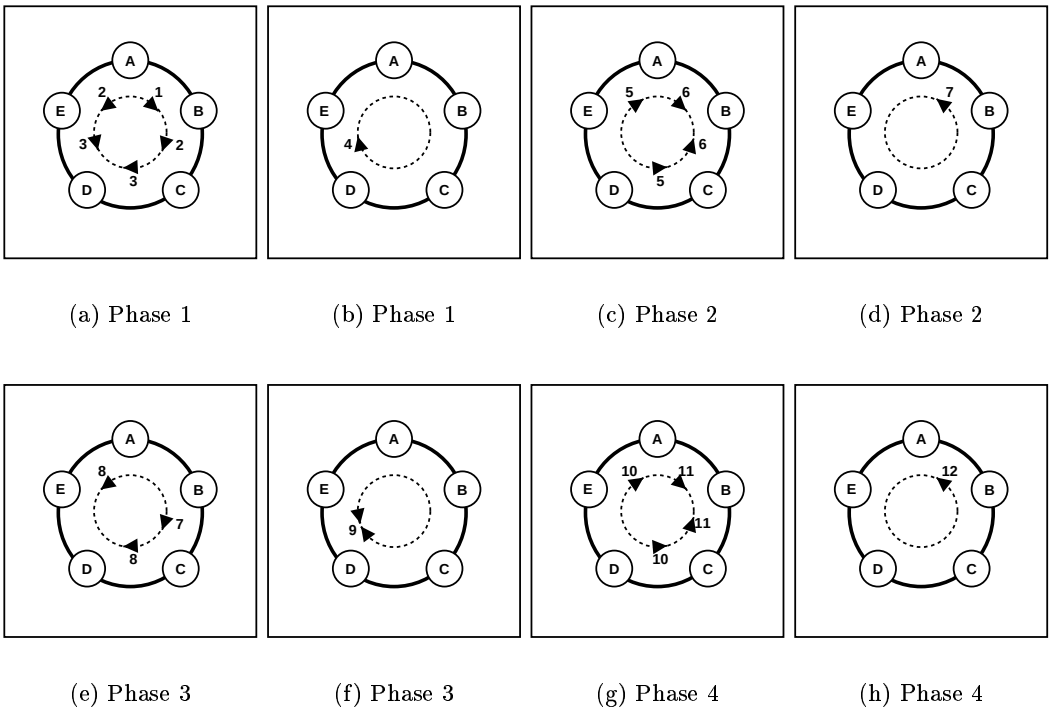
2.5.2 Dependent Reliable Broadcasts

Dependent Reliable Broadcasts in a bidirectional ring architecture can also be interleaved like dependent Reliable Broadcasts in an unidirectional ring architecture, since every member has the same knowledge. If two Reliable Broadcasts depend on each other, the phase one of the second Reliable Broadcast is the phase two of the first Reliable Broadcast. The two members which perform the transition from phase one to two of the first Reliable Broadcast inject the phase one transmission of the second Reliable Broadcast. Since every member knows that a second Reliable Broadcast is caused by the first Reliable Broadcast, the phase one transmissions of the second Reliable Broadcast are replacing the phase two transmissions of the first Reliable Broadcast.

In such a three-phase Reliable Broadcast, every member stores an incoming phase one or two transmission in a list and removes it if the related phase three transmission is received. So that a phase one transmission is forwarded if it was not in the list before, a phase two transmission is forwarded if the related phase one transmission is in the list and a phase three transmission is

forwarded if the related phase two transmission is in the list. A transition from phase one to two is performed always at two members if an incoming phase one transmission was in the list before and a transition from phase two to three is performed always at two members if an incoming phase two transmission was in the list before. The algorithm ends always at two members if the related phase one transmission to an incoming phase three transmission is not in the list. The phase one and two transmissions are reliably broadcasted similar to the phase one of a two-phase Reliable Broadcast.

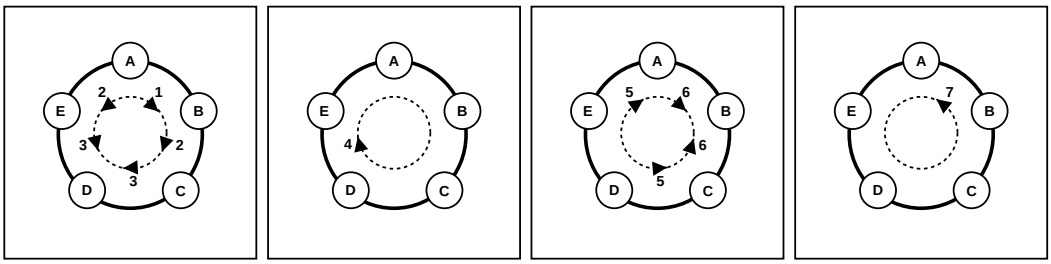
Two sequentialized Reliable Broadcasts in a bidirectional ring architecture perform $O(n) \approx 2n$ operations with $C(n) \approx 4n$ communications, where the maximum number of simultaneous communications is 2. But two interleaved Reliable Broadcasts in an unidirectional ring architecture perform $O(n) \approx n + n/2$ operations with $C(n) = 3n$ communications, where the maximum number of simultaneous communications is 2. Since $O(n)$ and $C(n)$ are linear dependent on n and the number of simultaneous communications is not dependent on n, both algorithms for dependent Reliable Broadcasts in a bidirectional ring architecture are linear scalable. But the interleaved Reliable Broadcast is more efficient than the sequentialized Reliable Broadcast in saving $n/2$ operation counts and n communications, which is linear to the number of members.



(a) Phase 1 (b) Phase 1 (c) Phase 2 (d) Phase 2

(e) Phase 3 (f) Phase 3 (g) Phase 4 (h) Phase 4

Figure 2.28: Sequentialized Bidirectional Reliable Broadcasts ($O(n) \approx 2n$, $C(n) \approx 4n$)



(a) Phase 1 (b) Phase 1 (c) Phase 2 (d) Phase 2

(e) Phase 3 (f) Phase 3

Figure 2.29: Interleaved Bidirectional Reliable Broadcasts ($O(n) \approx n + n/2$, $C(n) \approx 3n$)

2.5.3 Member Fault Recovery

Since member faults may occur, the Reliable Broadcast in a bidirectional ring architecture needs to recover from them in a fault-tolerant distributed system. When a member fault is recognized by the communication system, the two neighbor members of the faulty member connect to each other. Since transmissions are sent in both directions in the ring, both neighbor members had sent and received transmissions to and from the faulty member and must assume that all transmissions of the current phase are lost. They must be sent again, because previous phase transmissions are always reliably broadcasted. Both neighbor members are able to filter already received transmissions, because they know which transmissions were received before the member fault. Every transmission which is sent again has additionally a recovery flag, so that phase transitions are performed only once.

To avoid illegal states of the Reliable Broadcast algorithm e.g. phase transitions at only one member, every member needs to save additionally the information which transmission was received from which direction in the ring to be able to filter only from the faulty direction already received transmissions. Furthermore, due to the bidirectional information transport of the Reliable Broadcast algorithm in a bidirectional ring architecture, both sides together have the complete knowledge of the ring which went not lost during the member fault i.e. all lost transmissions expected to be received by the other side. This includes also the last-phase transmissions which are sent, went lost and are still pending on the other side. So that all current-phase transmissions are sent again without any additional recovery message.

Last-phase transmissions are generated and sent back to the other side if the related previous-phase transmission were received from this side during a recovery i.e. with a recovery flag. More clearly, a member which has sent a phase one transmission to a faulty member before the fault assumes that this transmission went lost, if no phase one or two transmission was received from the faulty member before the fault. So that it is sent again during a recovery. If the member on other side of the faulty member sends a phase one transmission to the faulty member before the fault and no phase one or

two transmission was received from the faulty member before the fault, it assumes that this phase one transmission went lost. The phase one transmission is sent again, both transmissions cross and the phase transition is executed correctly.

But, if the member on other side of the faulty member had received the phase one transmission from the faulty member before the fault, and if this member has received a phase one or two transmission from the other neighbor and sent a phase two transmission to the faulty member before the fault which was not received by the member on the other side of the faulty member, so that a phase two transmission went lost. The phase one transmission is sent again by the member which is waiting for the lost phase two transmission and the other member receives a phase one transmission related to a reliably broadcasted phase one transmission. In this case, the phase two transmission is generated and sent back to the other side, since the other member expects this phase two transmission which previously went lost.

2.5.4 Multiple and Cascaded Member Faults Recovery

Every section of faulty members in a bidirectional ring architecture is managed by the member fault recovery algorithm like a single member fault. The correct members are connected to each other and start the member fault recovery. Since the single member fault recovery algorithm assumes that all current-phase transmissions are lost and must be sent again, lost last-phase transmissions are also sent again and illegal states of the Reliable Broadcast algorithm which occur due to congestion in an asynchronous bidirectional ring architecture are corrected, the recovery for a section of faulty members in a bidirectional ring architecture works correctly and completely. Cascaded member faults in a bidirectional ring architecture are handled like a single or multiple member fault for every during a recovery cycle occurring single or multiple member fault.

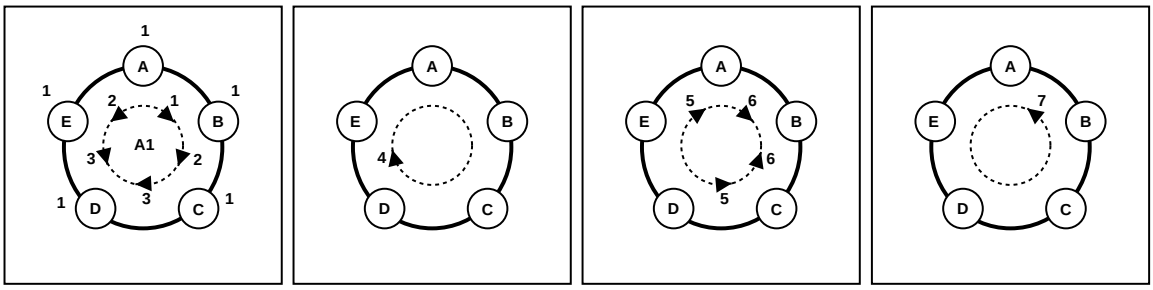
2.5.5 Atomic Broadcast

The Atomic Broadcast in a bidirectional ring architecture extends the Reliable Broadcast in a bidirectional ring architecture by numbering algorithm and sorting technique used in the Atomic Broadcast in an unidirectional ring. Every member has a local copy of the global number which is increased if a new Atomic Broadcast is started. The local copy is updated with higher received values. Reliably broadcasted phase one transmissions are ordered by the number, and by the priority on equal numbers. They are sorted at each member by insertion into a sorted list on receiving. So that a reliably broadcasted phase one transmission is atomically broadcasted if all previous phase one transmissions are reliably broadcasted without any blocking, which provides a correct, portable and heterogeneous algorithm since time stamps are not used and phase one transmission blocking is avoided.

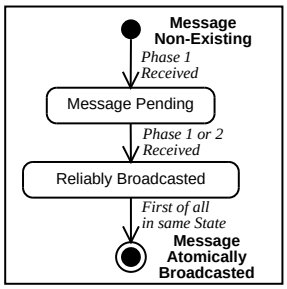
The local copy of the global number is increased and the transmission id with number and member priority is created before a phase one of the Atomic Broadcast is injected. In phase one, a member sends a phase one transmission containing the transmission id to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase one to phase two which is always executed at two members. In phase two, a member sends a phase two transmission containing the transmission id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Atomic Broadcast always at two members and each member knows that every member knows the phase one transmission and its globally unique order.

A phase one transmission is atomically broadcasted and both transmissions are removed from the list if no phase one transmission with lower number or equal number but lower member priority is pending i.e. not reliably broadcasted. Operation count, communication costs and fault recovery are similar to the Reliable Broadcast group communication service. The operation count is extended by

the waiting for previous phase one transmissions to be reliably broadcasted which may be different at every member and with every Atomic Broadcast.



(a) Phase 1 (b) Phase 1 (c) Phase 2 (d) Phase 2



(e) State-Machine

Figure 2.30: Atomic Broadcast in a Bidirectional Ring ($O(n) \approx n$, $C(n) \approx 2n$)

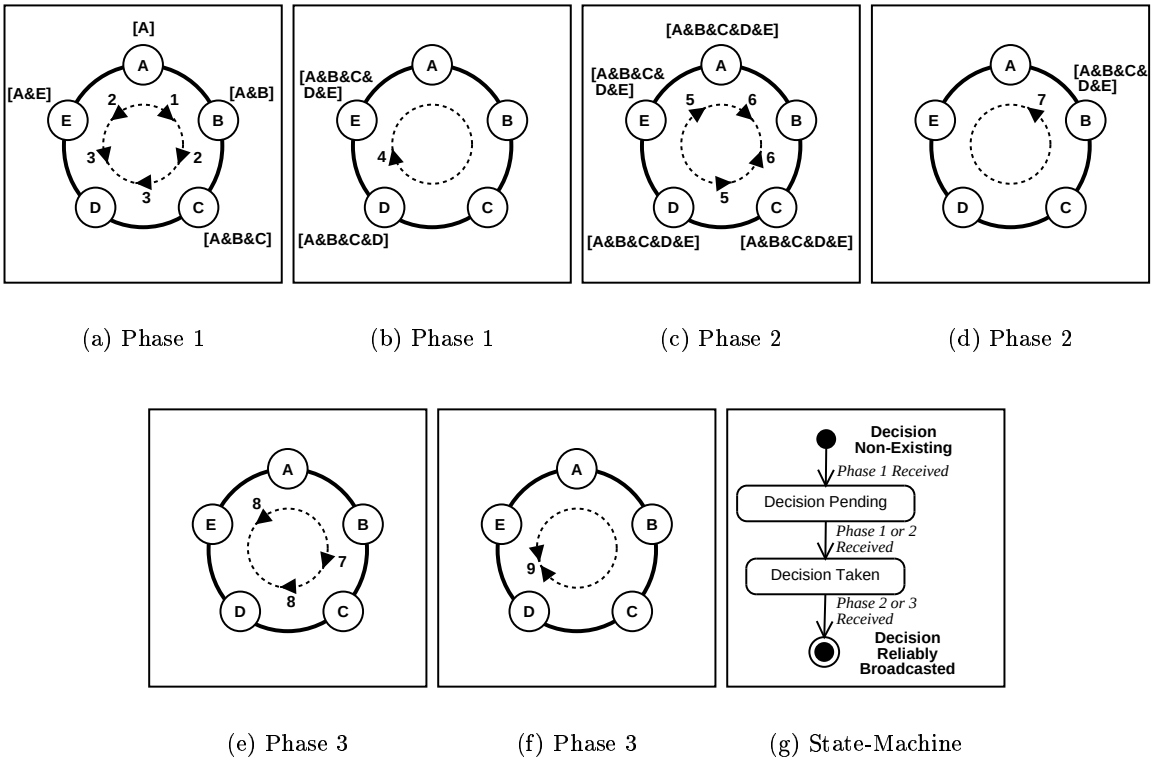
2.5.6 Distributed Agreement

The Distributed Agreement group communication service in a bidirectional ring architecture is based on two interleaved Reliable Broadcasts in a bidirectional ring architecture similar to the Distributed Agreement in an unidirectional ring architecture. Collective communication is used to avoid n-times Reliable Broadcasts, so that one Reliable Broadcast is executed to collect the decision from all members and a second Reliable Broadcast is executed to reliably broadcast the final result. Two interleaved Reliable Broadcasts are used to save $n/2$ operation counts, where the voting algorithm is realized similar to the Distributed Agreement in an unidirectional ring architecture.

The regeneration of a phase two transmission in a recovery needs to know the final decision. This is possible if every member stores the received final decision locally until a phase three transmission is received. So that in phase one, a member sends a phase one transmission containing the combined decision to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase one to phase two and is performed always at two members. In phase two, a member sends a phase two transmission containing final decision to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase two to three always at two members.

In phase three, a member sends a phase three transmission containing an acknowledgement to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Distributed Agreement always at two members and each member knows that every member knows the final decision. The Distributed Agreement in a bidirectional ring architecture

performs $O(n) \approx n+n/2$ operations with $C(n) \approx 3n$ communications, where the maximum number of simultaneous communications is 2. The transmission size does not depend on the number of members in the ring due to the use of collective communication. So that new members are able to be involved in already running Distributed Agreements, because the algorithm is scalable at any time.



(a) Phase 1 (b) Phase 1 (c) Phase 2 (d) Phase 2

(e) Phase 3 (f) Phase 3 (g) State-Machine

Figure 2.31: Distributed Agreement in a Bidirectional Ring ($O(n) \approx n + n/2$, $C(n) \approx 3n$)

2.5.7 Transaction Control

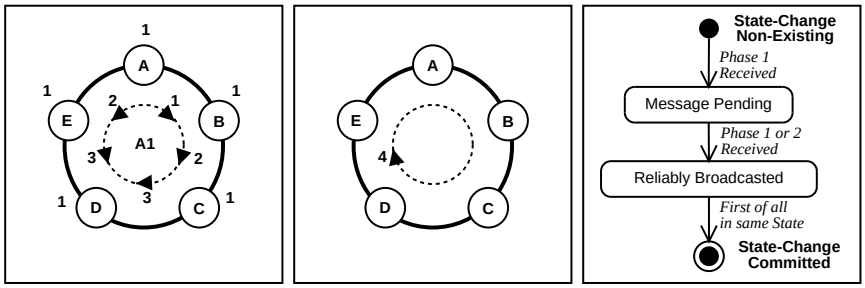
The Transaction Control group communication service in a bidirectional ring architecture is based on the Atomic Broadcast in a bidirectional ring architecture and may involve additionally the Reliable Broadcast or the Distributed Agreement similar to the unidirectional ring architecture. The Transaction Control involves the Reliable Broadcast or the Distributed Agreement to ensure a correct transaction commit or rejection based on the transaction execution result for active state changes which are based on non-replicated data. The Atomic Broadcast also ensures a globally unique order of a followed Reliable Broadcast or Distributed Agreement, since transmissions cannot pass each other in a ring. The Transaction Control in a bidirectional ring architecture is always linear scalable, since all used group communication services are linear scalable.

Simple Transaction Control

A distributed system which supports inactive state changes which are based only on replicated data executes an Atomic Broadcast of a transaction in a bidirectional ring architecture. A transaction is executed if it is atomically broadcasted and committed if the transaction execution result is positive. The execution of one state change performs $O(n) \approx n$ operations with $C(n) \approx 2n$ communications, where the maximum number of simultaneous communications is 2. Such a distributed system executes and commits illegal transactions when they are based on non-replicated data and revert them using

state changes, so that the global state is consistently replicated but not always correct. In phase one, a member sends a phase one transmission to its neighbor in the ring containing the state change and the transaction id until it is received by a member which has it already received before.

This condition triggers the transition from phase one to phase two and is performed always at two members. In phase two, a member sends a phase two transmission related to the phase one transmission containing the transaction id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Transaction Control always at two members and each member executes and commits or rejects the state change based on the local execution result if it is atomically broadcasted.



(a) Phase 1 (b) Phase 2 (c) State-Machine

Figure 2.32: Simple Transaction Control in a Bidirectional Ring ($O(n) \approx n$, $C(n) \approx 2n$)

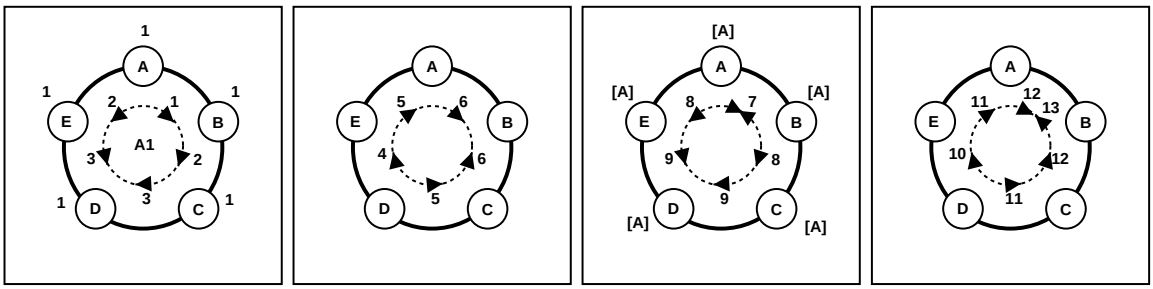
Transaction Control with Reliable Broadcast

A distributed system which supports active state changes based on non-replicated data of only one server executes an Atomic Broadcast of a transaction in a bidirectional ring architecture followed by a Reliable Broadcast in an bidirectional ring architecture of the transaction execution result performed by the server with the used non-replicated data. A transaction is executed if it is atomically broadcasted, committed if the transaction execution result is reliably broadcasted and positive, and rejected if the transaction execution result is reliably broadcasted but negative. Illegal transactions are executed, committed and reverted when they are based on non-replicated data of more than one members. The Atomic Broadcast and the Reliable Broadcast cannot be interleaved, since the member which works with non-replicated data may not always the same.

The execution of one state change performs $O(n) \approx 2n$ operations with $C(n) \approx 4n$ communications, where the maximum number of simultaneous communications is 2. In phase one, a member sends a phase one transmission to its neighbor in the ring containing the state change and the transaction id until it is received by a member which has it already received before. This condition triggers the transition from phase one to phase two and is performed always at two members. In phase two, a member sends a phase two transmission related to the phase one transmission containing the transaction id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Atomic Broadcast always at two members, and each member executes the state change if it is atomically broadcasted.

The Reliable Broadcast is injected by the member which executed the state change based on non-replicated data, where the Reliable Broadcast begins with phase three. In phase three, a member sends a phase three transmission related to the phase two transmission containing the transaction id and the transaction execution result to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase three to phase four and is performed always at two members. The state change is committed or rejected based

on the in phase three received execution result during phase four. In phase four, a member sends a phase four transmission related to the phase three transmission containing the transaction id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Transaction Control always at two members and each member has executed and committed or rejected the state change.



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

Figure 2.33: Transaction Control with Reliable Broadcast ($O(n) \approx 2n$, $C(n) \approx 4n$)

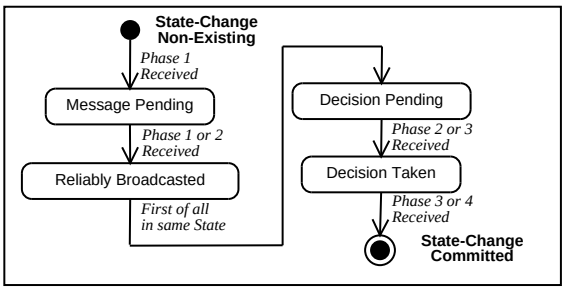


Figure 2.34: State-Machine of Transaction Control with Reliable Broadcast

Transaction Control with Distributed Agreement

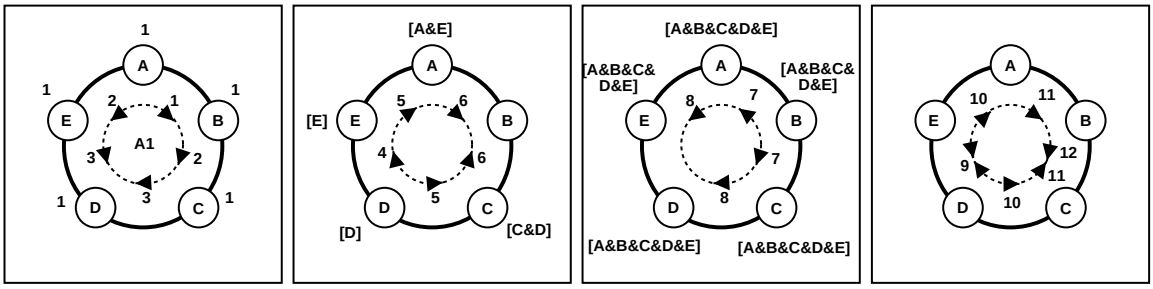
A distributed system which supports inactive and active state changes based on replicated data or non-replicated data of one or more servers executes an Atomic Broadcast of a transaction followed by a Distributed Agreement on the execution result in a bidirectional ring architecture. A transaction is executed if it is atomically broadcasted, committed if all members agree on the execution result and rejected otherwise. The voting algorithm may depend on the transaction type. The Atomic Broadcast and the Distributed Agreement can always be interleaved if the second phase of the Atomic Broadcast is blocked during the transition from phase one to two until the transaction is atomically broadcasted i.e. all previous transactions are unblocked, so that a transaction is executed and its result is collected during phase two of the interleaved algorithm.

The execution of one state change performs $O(n) \approx 2n$, operations with $C(n) \approx 4n$ communications, where the maximum number of simultaneous communications is 2. In phase one, a member sends a phase one transmission to its neighbor in the ring containing the state change and the transaction id until it is received by a member which has it already received before. The transition from phase one to two is performed always at two members if and only if no phase one transmission with lower number or equal number but lower priority is pending i.e. not reliably broadcasted. The phase one transmissions of all other state changes terminate. This is safe since the phase one was reliably

broadcasted and the communication blocking takes place in a second phase of a Reliable Broadcast i.e. all members have this state change in a blocked state and are waiting for the release. The state change is executed during phase two and its execution result is collected using collective communication.

Every member stores locally the result value sent to the next member. In phase two, a member sends a phase two transmission related to the phase one transmission containing the transaction id and the collected transaction execution result to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Atomic Broadcast always at two members and each member has executed the state change and the results are collected into the final result. In phase three, a member sends a phase three transmission related to the phase two transmission containing the transaction id and the final transaction execution result to its neighbor in the ring until it is received by a member which has it already received before. This condition triggers the transition from phase three to phase four and is performed always at two members.

The state change is committed or rejected based on the in phase three received execution result during phase four. In phase four, a member sends a phase four transmission related to the phase three transmission containing the transaction id to its neighbor in the ring until it is received by a member which has it already received before. This condition terminates the Transaction Control always at two members and each member has executed and committed or rejected the state change.



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

Figure 2.35: Transaction Control with Distributed Agreement ($O(n) \approx 2n$, $C(n) \approx 4n$)

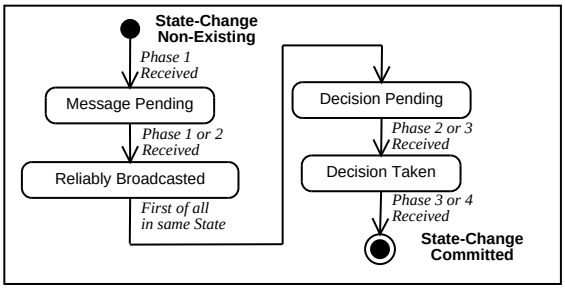


Figure 2.36: State-Machine of Transaction Control with Distributed Agreement

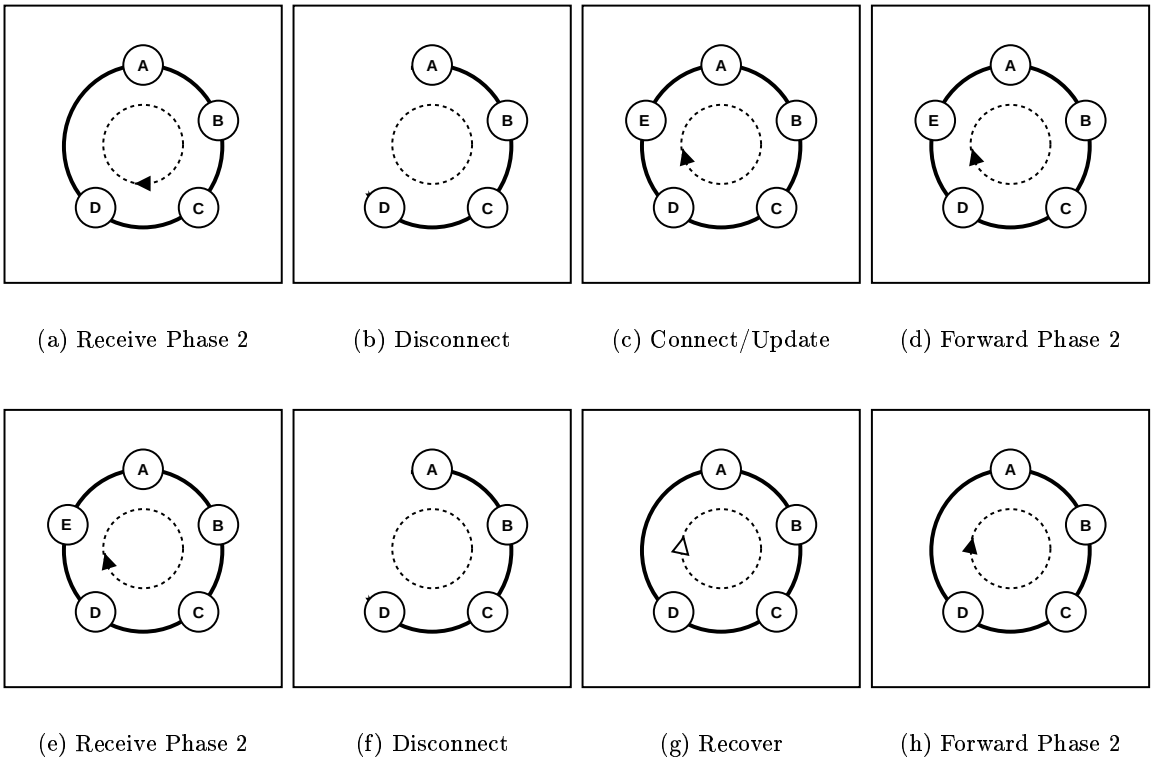
Since phase one transmissions appear to be atomically broadcasted in blocks, the transmission phases two, three and four are applied to such a block of atomically broadcasted phase one transmissions. So that a block of transactions is executed in the right order and a block of execution results is collected in phase two, and a block of execution results is reliably broadcasted with the phases three and four, where a block of transactions is committed in the right order in phase four. Such a block consists of simultaneously injected transactions only i.e. transactions with the same number. This block processing technique saves the time of sending x-1 messages for a block of x transactions by

sending a message which consists of x results and the id of the leading transaction.

2.5.8 Membership

A member is added to the server group using an 'admit member' state change similar to the unidirectional ring. During the execution of this state change in a bidirectional ring architecture, the dedicated previous member and the dedicated next member of the new member disconnect each other and try to connect the new member. The previous member sends its complete state to the new member on success and the member is added to the server group. The previous and the next member are reconnected and the member admission state change fails at all members if the connection to the new member cannot be established or the new member fails during the state update. Since a state change is executed and committed in different phases, all members maintain an extra local list of members which represents the current server state and which may be uncommitted.

Similar to the unidirectional ring, a member is removed using the controlled member fault during the execution of a 'remove member' state change which exactly defines the member to be removed. A member which recognizes that the next member has to be removed waits for a disconnect from this member. A member which recognizes that it has to be removed, it disconnects the links to the previous and the next member in the ring on state change execution. The next member falls automatically into the member fault recovery algorithm. The previous and the next member are reconnected to each other by the member fault recovery algorithm which sends also lost transmissions again and does not report a member fault, since the disconnect occurred during the 'remove member' state change execution and the member fault is always reported by the previous member.



(a) Receive Phase 2 (b) Disconnect (c) Connect/Update (d) Forward Phase 2

(e) Receive Phase 2 (f) Disconnect (g) Recover (h) Forward Phase 2

Figure 2.37: Member Admission (a-d) and Removal (e-h) in a Bidirectional Ring

3 Implementation Strategy

The distributed control of the Harness distributed virtual machine service manages the consistent state-replication in the hot- and warm-standby server groups, so that each member of the distributed virtual machine service runs the same distributed control algorithm in hot- and/or warm-standby mode. A prototype which runs the hot-standby mode only is realized due to the complexity of the designed distributed system architecture. The prototype allows to experiment with the hot-standby mode and its distributed state-replication algorithm to test the requirements of correctness, fault-tolerance, scalability, heterogeneity and efficiency in the real world of heterogeneous distributed computing. The implementation of the warm-standby mode can be derived from the prototype in the future, since it uses similar group communication services.

3.1 Requirements

The prototype allows to experiment with the consistent state-replication of the hot-standby mode of the Harness distributed control. The requirements for such a prototype do not force the implementation to fit in a very specific environment or to implement an interface which is needed by external software components. The main goal is to implement a prototype which runs the distributed control algorithm in a very flexible software architecture to allow experimental changes in the algorithm and data structures. Furthermore, the state-replication algorithm requires already a specific system behavior and design. So that only some major requirements which are defined additionally need to be fulfilled to enable the prototype to be extended to an implementation, which can be used in the Harness distributed virtual machine service in the future.

3.1.1 Programming Language

The development of the Harness distributed virtual machine is a collaborative effort between the Oak Ridge National Laboratory, the University of Tennessee, and the Emory University. The main goal of the Oak Ridge National Laboratory is to develop a C/C++ version of the Harness distributed virtual machine, where the University of Tennessee and the Emory University are working on a Java version. C or C++ is preferred for the prototype, since this work is part of the development at the Oak Ridge National Laboratory. Furthermore, the prototype implements group communication services to maintain the consistency of the distributed state-database based on TCP/IP communication. A native C/C++ implementation provides more performance due to the direct access to the operating system without any additional virtual machine.

On the other hand, the Harness distributed virtual machine is conceived for heterogeneous distributed systems. A Java implementation of the prototype supports every operating system which is supported by a Java virtual machine implementation. Additionally, the Java virtual machine provides heterogeneity by using its own data types which do not depend on the operating system or hardware architecture i.e. the bit length and byte order are maintained. So that a Java implementation of the distributed control is heterogeneous but not so efficient as a C/C++ implementation. A C/C++ implementation of the prototype which maintains unique data types with defined bit length and byte order for all transferred data is heterogeneous and efficient.

The main purpose of the prototype is to test a distributed state-replication algorithm. So it is implemented in C++ to provide efficiency, heterogeneity and flexibility. It uses operating system calls directly and maintains unique data types for transferred data. The flexibility is provided by an object-oriented design, where the final version of the implementation can always be optimized into a non-object-oriented C version if the algorithms are tested and changes caused by the test results are made. Template classes and the C++ standard template library are generally not used due to the non-existing support for template classes by compilers on some operating systems and on some hardware architectures. The data types are based on sequences of bytes, where bit-fields are not used to provide heterogeneity.

3.1.2 System Architecture

The analysis of the operation count of the algorithms developed in the preliminary system design show that a bidirectional connected ring of servers is always more efficient than an unidirectional connected ring. The prototype supports the bidirectional connected ring only, where a server mode of the prototype realizes a ring node and a client mode of the prototype realizes an external client which injects transaction requests to the on the local machine residing ring node. So that the prototype is running at each server twice, where one instance implements the ring node of the hot-standby server group which maintains the replication and the other instance is a replacement for the local distributed virtual machine service. The hot-standby mode of the prototype provides the interface to the local distributed virtual machine service replacement via a local TCP/IP server.

The hot-standby server group of the Harness distributed virtual machine uses active state changes which are based on non-replicated or replicated data of one or more servers to admit or remove members, to start or stop services, to report service state changes and to execute service state changes. The Transaction Control group communication service with Distributed Agreement is realized in the prototype to provide a state-replication algorithm for a consistent fault-tolerant distributed database with such active state changes. The warm-standby server group uses inactive state changes which are based on replicated data only, since the complete database backup is overwritten by the execution of one state change. So that the simple Transaction Control based on one Atomic Broadcast is needed. But this is not implemented in the prototype.

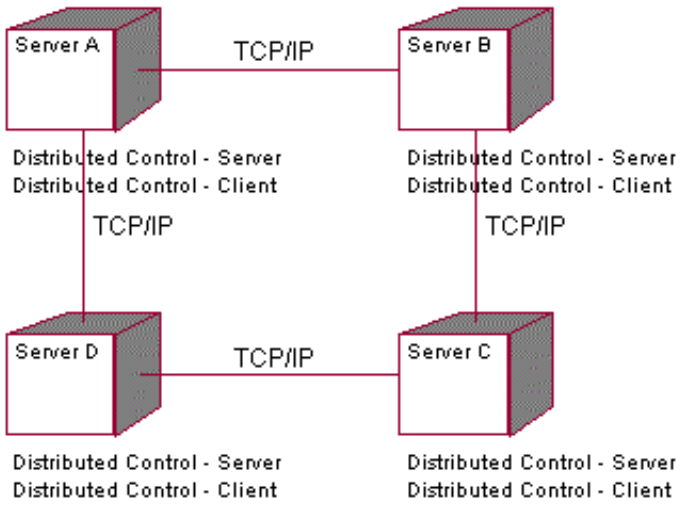


Figure 3.1: Distributed Control Prototype Deployment Diagram

3.1.3 Software Design

The object-oriented software design provides a general mechanism to heterogeneously transfer objects from one server to another over the network. A distributed object mechanism is not implemented to avoid remote method invocation. Transferable objects can be packed and unpacked into or from a byte oriented buffer where bit length and byte order of data types are maintained. The data of pending transactions is stored at each server locally and accessed frequently, and the number of simultaneous transactions in a system with n servers which are allowed to inject x transactions simultaneously is xn. At the moment, a server is allowed to have only one transaction in the system, but future versions may allow more. Since transactions have to be ordered into a sorted list and their order is known, a balanced binary tree is used to provide $log(n)$ operation counts.

3.1.4 System Environment

The implementation of the prototype uses System V socket communication to communicate with other instances in the bidirectional server ring and to communicate between a ring server instance and the local client instance. Every server system must provide System V sockets, where compatibility libraries may be used. Additionally, the TCP/IP network protocol must be provided by the operating system of a server and all servers must be connected in a TCP/IP network. An ANSI-C/ANSI-C++ compiler and the ANSI-C libraries are needed to compile the source code on a system, where the C++ standard template library and C++ template classes are not used. The amount of used memory and the disk usage of the prototype highly depends on the amount of replicated data. The tests require at least 8 MB by the application usable RAM and at least 20MB disk space.

The development environment is a set of IBM-compatible workstations with Intel-compatible processors, more than 32MB RAM and 20MB free user disk space. The workstations are connected via a TCP/IP network. The prototype is developed under the Linux operating system (Mandrake 7.2 Distribution) using the make tool and the C++ compiler. The testing environment for the correctness and fault-tolerance requirements is the development environment. During the tests, the client and server processes are started via the command line user interface of every physically server without using any additional parallel computing environment, such as PVM, to provide a more flexible test environment. Both, server process and client process, can be interrupted i.e. killed at any time during a test by using a by the keyboard caused signal i.e. using Ctrl.+C.

3.2 Testing Techniques

The test of the prototype consists of three different phases. The distributed state-replication algorithm is partially tested without an implementation in a dry run test. Major software components are tested in the component test and the distributed state-replication algorithm tested in the system test. Since a prototype is implemented to experiment with a distributed state-replication algorithm, the requirements for the system test grow with the implemented capabilities and may change if unexpected system behavior leads to new discoveries e.g. undesirable side-effects or non-linear scalability. A step-by-step testing strategy is performed during the system test to gather experience with the distributed system behavior due to the complexity of the distributed state-replication algorithm. Negative test results may cause implementation changes if the reason for a failed test can be found.

The tests have generally an exploratory character to earn experience and discover existing problems. This may include also a review of the theoretical system design if any problem occurs to provide a system analysis based on experimental results. Such a system analysis may lead to major system design changes or to impossibility proofs. So that the test results and conclusions prepare the basis for a complete review and system analysis based on experimental results.

3.2.1 Dry Run Test

The test of the Transaction Control Algorithm is restricted to the test of the Atomic Broadcast of a transaction in a bidirectional ring to show that the transaction ordering works. The complete Transaction Control Algorithm is not tested, since the descriptions of all in the preliminary system design illustrated algorithms already imply test results made during the algorithm development. Furthermore, a complete dry run test is hard to execute due to the complexity of the developed distributed algorithms. Such a test needs to keep in mind all assumptions which are made for each system state, where the set of possible system states of the Transaction Control Algorithm with Distributed Agreement is huge.

3.2.2 Component Test

The component test of the prototype evaluates single software components. This is necessary due to the complexity of the implementation and avoids single component failures in the system test. It provides a more completely software test, since the system test does not test all software components. The component test checks major components and mechanisms such as the object transfer mechanism, the balanced binary tree and the non-blocking communication. The object transfer mechanism must correctly transfer objects, so that objects of the same class with the same members are created. The balanced binary tree must provide correct tree modification algorithms, so that the tree is maintained correctly. The non-blocking communication must provide a non-blocking reading from sockets if no data is available and blocking reading from sockets until a complete transmission was received.

3.2.3 System Test

The system test of the prototype evaluates the correctness of the implemented state-replication algorithm with one, two or more than three servers. The algorithm shows in theory a different behavior in these three cases. One server must emulate the ring communication locally or the admit state change cannot be applied, because it expects a logical ring of servers with a dedicated previous and a dedicated next member, which is the one server itself. The local state is completely copied to the new member which starts its own state-replication algorithm after the state change execution. In the ring emulation, a server forwards transmissions to itself locally without using network communication, where phase changes occur only on one server and immediately after processing one phase.

Two or more servers operate in a server ring. Congestion and illegal states of the Reliable Broadcast algorithm which are corrected often occur in a two server group due to the small amount of members in the ring and the resulting small round trip delay. The transmissions are not faster in one direction than in the other direction. But one direction is automatically preferred, since a server can only send in one direction at a time. The two server ring is not a sufficient test configuration for member faults, since all members in the ring recognize a member fault and a member fault falls back into the one server group. In a more than three server group, one member is always not connected to at least one other member, so that a real asynchronous connected bidirectional ring is formed.

The prototype must correctly execute and commit a member admission if the new member expects the admission, the position of the new member is correct and the new member does not fail. The member admission fails otherwise. The prototype must correctly execute and commit a member removal if the server to be removed is a member of the server group. The prototype must correctly execute a member fault recovery and commit a member fault state change if a member in the ring fails. The prototype must correctly execute a print transaction which appends the transaction id and the message of the print transaction to a file with the IP address of the server as filename. All members (servers) execute the same state changes, so that their files have the same content despite synchronous transactions. Additionally, such synchronous transactions are shared fairly.

4 Detailed Software Design

The prototype is developed in C++ on Linux [31] and documented in UML [29]. An object-oriented architecture provides a flexible software design, where several objects encapsulate services which are needed by the distributed state-replication algorithm for a consistent fault-tolerant distributed database. The by the state-replication algorithm used temporary data e.g. transactions and transmissions is stored dynamically. Such a dynamic memory allocation slows down the performance and has limits defined by the operating system. But an implementation of a static memory usage is much more complex and moves the software component optimization before the aspired state-replication algorithm test. Objects are created at server or client start and at message receiving time. They are destroyed on server or client shutdown or if they are not used any more.

The client and the server run only one process. The client connects to the server, produces transactions, injects transaction requests and disconnects. The server runs the state-replication algorithm, where transaction requests are received from the client and the transactions are injected when allowed. A queue stores incoming transaction requests until they can be injected to provide a more asynchronous connection between client and server. The server processes transmissions fairly in one process to avoid interprocess communication between multiple processes and synchronization of the mutual exclusive state-replication algorithm including the list of transactions and related transmissions. The transmission type of the last received and last sent transmission only is stored for both directions per transaction to save memory.

Since the transaction, its execution result and the last sent and received transmission types are stored locally. Every lost transmission which is not a last-phase transmission can be regenerated and received transmissions can be filtered. Last-phase transmissions are generated and sent back to the sender if transmissions are related to recently deleted transactions. The transaction numbering algorithm provides the distinction between recently deleted and unknown transactions despite wrap-around order. The Transaction Control algorithm with Distributed Agreement is implemented in a modified version by moving the transaction commit or rejection from phase four to phase three. This provides a less latency of transactions, where the algorithmic semantic is still maintained. A transaction is committed or rejected when the final result is known instead of when it is reliably broadcasted.

4.1 Object Model

Several different objects enable the prototype to run the distributed state-replication algorithm for a consistent fault-tolerant distributed state-database based on the Transaction Control group communication service with Distributed Agreement in a bidirectional ring architecture. These objects provide services to a distributed control object which implements the state-replication algorithm. A linked list stores objects into a list using a wide variety of list operations while a red-black tree realizes the balanced binary tree with $log(n)$ operation counts. The object transfer mechanism ensures a correct transfer of objects over the network, so that objects of the same class with the same members are created on the other side. This mechanism is used to transfer transmission objects which may contain a transaction object or a linked list object containing transaction execution result objects.

4.1.1 Linked List

The distributed control object and the database object need to store the list of members, where the distributed control object stores the current state and the database object the committed state. Additionally, transaction injection requests which are sent from the client to the server are stored into an incoming transmission queue object to allow a more asynchronous interaction between the clients and the server ring. An object instance of the LinkedList class provides a wide range of list access methods of a linked list, where objects of the super class Object are stored. The LinkedList class implements also a list iterator which points to a current list entry and can be moved to the previous, next, first or last list entry. Objects can be inserted into the list before or after the current list entry. The current list entry or all stored objects can be removed or deleted.

A node object stores the links to the previous and next node relative to its position in the list and the link to the stored object. Since the link to an object is stored and not the object itself, the linked list object can search for an object by comparing stored links with a given link to seek the list iterator to a known object. The link storing technique requires that objects are created before they are added to the list and deleted after they are removed from the list by the user i.e. by the object which uses the linked list. So that the destructor of the LinkedList class removes all objects from the list. But the objects are not destroyed. This is used to handle temporary data related to transactions more efficiently by creating objects only once and by forwarding its link from one object to another inside of one process. This avoids the copy operation via copy constructor when data is moved.

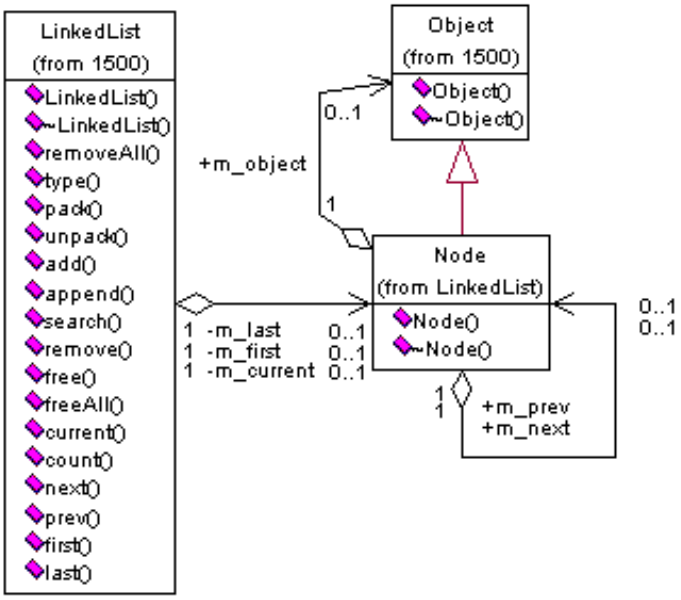


Figure 4.1: Linked List Class Diagram

4.1.2 Red-Black Tree

The distributed control object needs to store transactions and the related data efficiently. A red-black tree is basically a balanced binary tree and can be used to achieve $log(n)$ operation counts for access methods [28]. Since the order of incoming transactions is known and unique, transactions can be sorted by insertion into a red-black tree. A solution which is generally applicable is implemented by the RedBlackTree class. It stores objects of the super class Object associated to key objects of the super class Comparable. The Comparable class has abstract methods only, which are used to determine the key order i.e. if a key object is less than or equal to another. A method to determine whether a key object is greater than another is not implemented to avoid illegal definitions of the key order e.g. a key object is less than and greater than another.

A node object stores the links to the left, to the right and to the parent node relative to its position in the tree and the links to the stored object and key. The link storing technique requires the same restrictions for insert and remove operations as in the LinkedList class. A list iterator is implemented to provide sequential access to key objects and stored objects. A stored object can be retrieved with a key which fulfills the equal operation to the stored key. A red black tree object is able to store objects in a wrap-around order as long as the key operations are always correct. Such a wrap-around order exists if key value operations include overflow and underflow of key values. So that a key object with a key value which is the predecessor of a key value of another key object is the successor of this other key object. The key values form a ring of numbers.

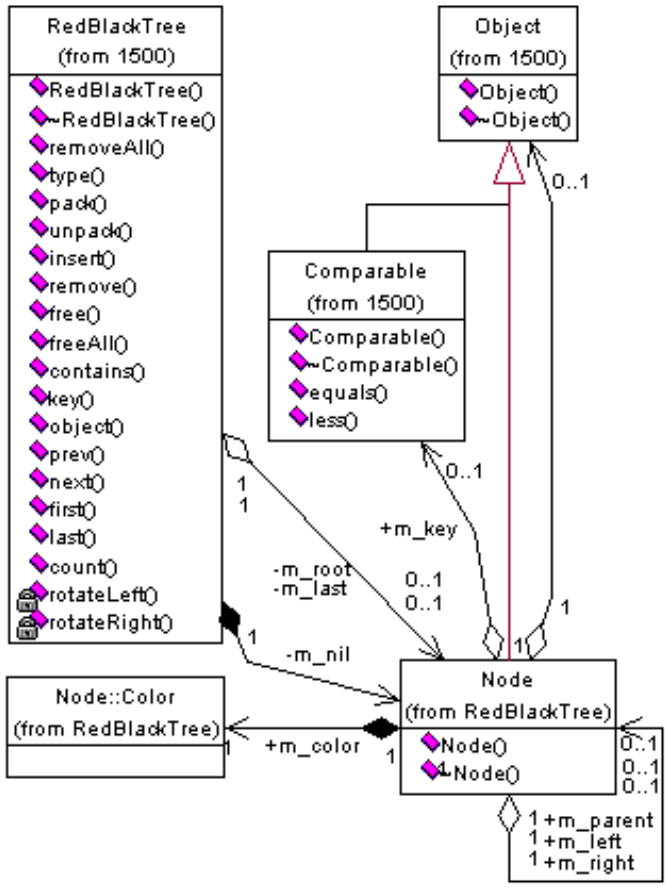


Figure 4.2: Red-Black Tree Class Diagram

4.1.3 Transferable Objects

The object transfer mechanism is based on the Transferable class. Every class which instantiates a transferable object needs to be derived from the Transferable class and needs to implement abstract methods of the Transferable class. One abstract method packs the transferable object into a message buffer and is called before sending. Another abstract method unpacks the transferable object from a message buffer and is called after receiving. The Transferable class implements also static methods for packing and unpacking of data to maintain unique data types with defined bit length and byte order for all data transferred in transmissions over the network. These static methods are used by transferable objects to pack and unpack members with basic data types. Member objects need to be transferable too, so that a transferable object contains only transferable member objects.

The Transferable class implements also the factory design pattern [30] by providing static pack and unpack methods for transferable objects. One method packs a given object of the Transferable class into a given buffer and the other method unpacks a new created object of the Transferable class from a given buffer, where the pack and unpack implementations of the transferable objects are used. Every of the Transferable class derived class needs to implement an additional abstract method to return a class type identifier for the object factory. This class type identifier is packed and transferred to enable the unpack method to create an object of the same class. The factory is used before sending and after receiving transferable objects by the sender, receiver and by transferable objects which contain transferable objects to pack and unpack such transferable member objects.

Classes of transferable objects are: Member, Result, Transaction, Transmission, LinkedList and RedBlackTree. This enables the distributed control object to send transmission objects over the network which may contain a transaction object, a linked list object with transaction execution result objects, a linked list object with member objects and a red-black tree object with transaction list entry objects. The object transfer mechanism is extendable and provides heterogeneity.

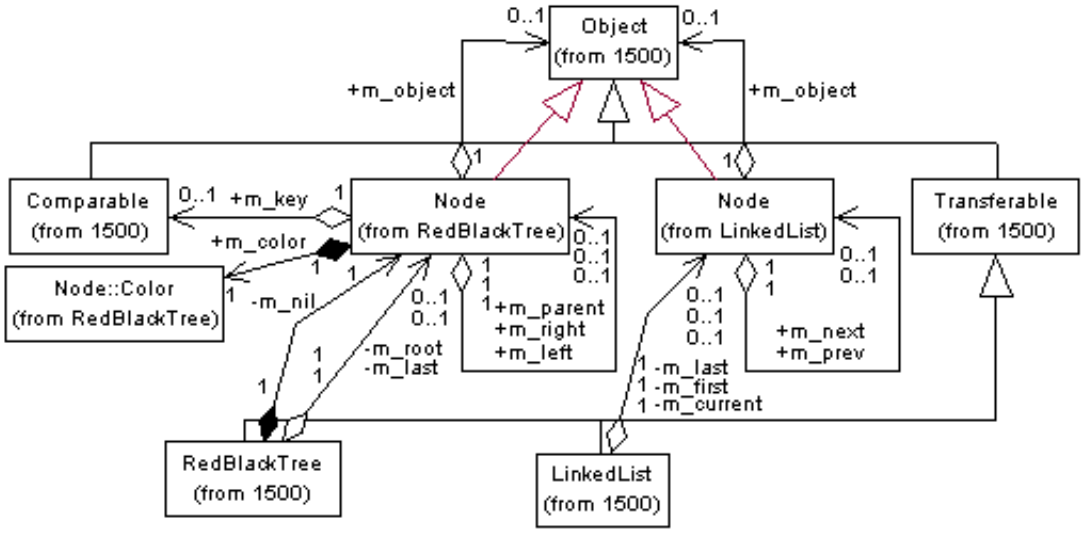
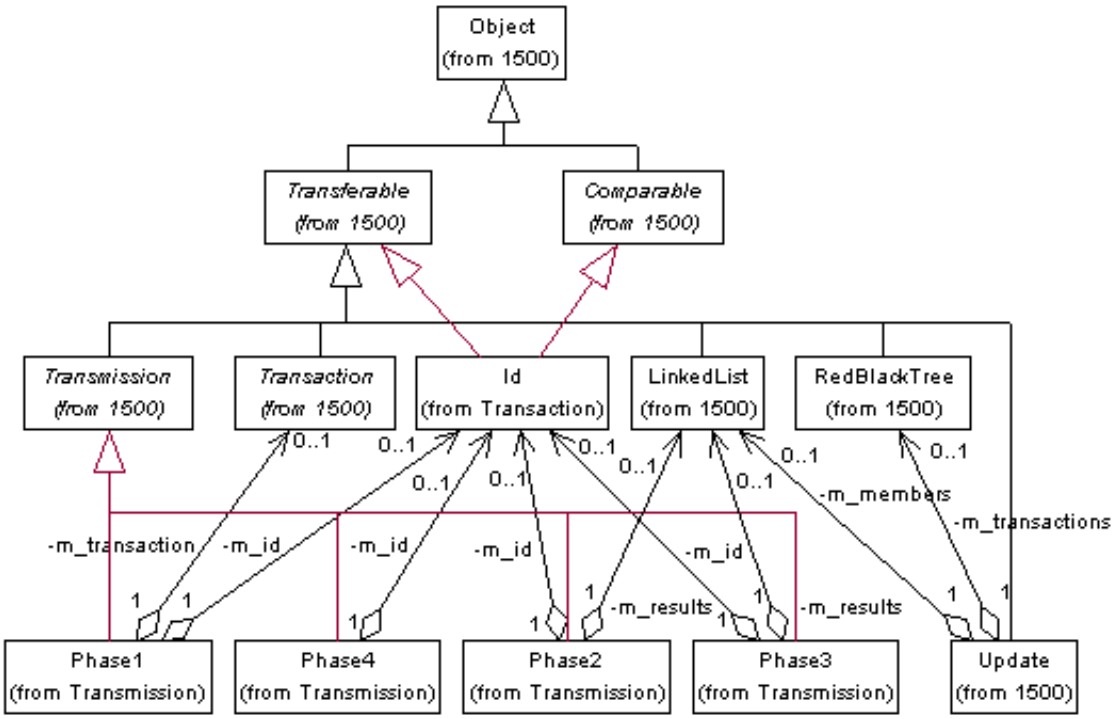


Figure 4.3: Transferable Linked List and Red-Black Tree Class Diagram

4.1.4 Transmissions

The distributed state-replication algorithm for a consistent fault-tolerant distributed state-database based on the Transaction Control group communication service with Distributed Agreement requires to send transmissions from one member to another in a bidirectional connected ring. Transmissions are transferable objects which contain objects depending on the transmission phase. The transmission classes Phase1, Phase2, Phase3 and Phase4 implement the necessary transmission phases. Additionally, an update transmission object of the Update class is used during a member admission to update the state of the new member. Transmissions of the state-replication algorithm contain always a transaction id which is used to identify the transaction related to a transmission.

The phase one transmission additionally contains the transaction object and phase two and three transmissions additionally contain the linked list object with the transaction execution result objects. Recovery transmissions of the phases one, two and three are sent during a member fault recovery. Since these transmissions have the same content as the lost transmissions but a different transmission type i.e. different class type. The classes for the recovery transmissions are directly derived from the transmission classes and override the class type method only. Transaction objects are sent from the client to the server as injection request without any covering transmission object, since a transaction is always a transferable object.



(a) Class Hierarchy

Figure 4.4: Transmission Class Diagram

4.1.5 Transactions

A transaction object encapsulates all data and algorithms of a state-change. The Transaction class is the base class of all transactions. It has abstract methods which must be implemented by derived transaction classes to provide the transaction execution, commit and rejection mechanisms called by the distributed control object. The Transaction class is derived from the Transferable class to enable transaction objects to be transferred over the network. The prototype implements a Print transaction class for test purpose, a Fault transaction class to report member faults and an Add and a Remove transaction class to manage the membership. The Fault transaction class has a member object of the LinkedList class to list the detected faulty members. A transaction object is always related to a transaction id object which contains the transaction number and the member priority.

The transaction number is used in a wrap-around order. It is increased before every phase one transmission injection, where the value may overflow. A globally unique order is maintained as long as predecessor and successor can be identified using subtraction of two values and casting into the transaction number data type. This requires $2t + 1$ possible transaction numbers, where t is the maximum number of simultaneous transactions in the system. A ring with n members where each of them is able to inject x transactions produces $t = xn$ simultaneous transactions. So that $2xn + 1$ possible transaction numbers are necessary. The prototype allows only one transaction per member at a time. But the transaction number is a 16 bit value which allows 32767 simultaneous transactions to avoid any restrictions during experiments with the state-replication algorithm.

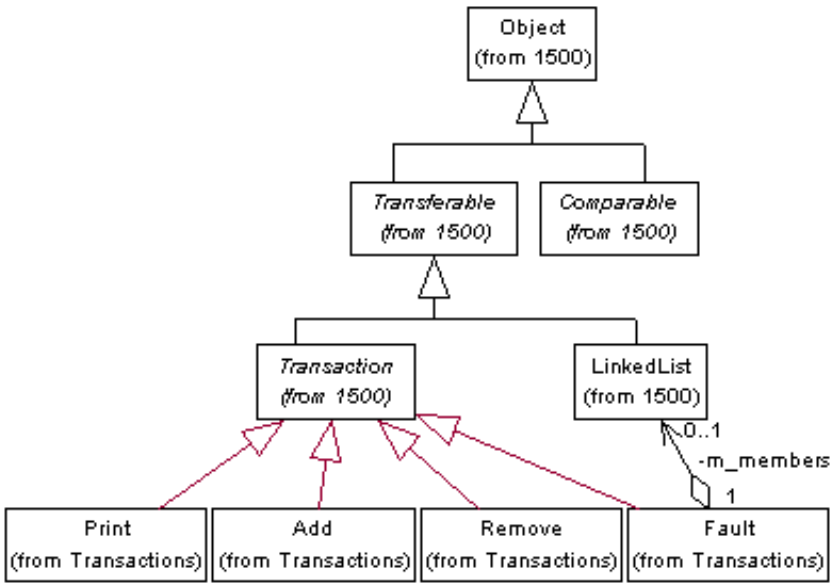


Figure 4.5: Transaction Classes

4.1.6 Distributed Control

The distributed control object is the main part of the prototype and runs the distributed state-replication algorithm for a consistent fault-tolerant distributed state-database based on the Transaction Control group communication service with Distributed Agreement in a bidirectional ring architecture. It controls one database object and three connection objects which represent the connections to the left member, to the right member and to the local client. The connection objects use client or server sockets with non-blocking communication depending on the direction in the ring. The distributed control object uses a red-black tree object to temporarily store transactions and a linked list object to store its own list of members which may not be committed and may differ to the list of members maintained in the database object.

The distributed control object runs the hot-standby mode only in the server mode of the prototype and uses methods to execute phase transitions (phase*to*) and reversed recovery phase transitions (phase*to*reversed). It implements the member fault recovery, where previously sent transmissions are sent again. The Database class is derived from the Transferable class to enable the database to be transferred to a new member as part of the update transmission. A transaction object involves the distributed control object and the database object during execution and commit, where the execution reads data from the database and changes data only in the distributed control object. Such changes are reverted at the distributed control object if a transaction is rejected.

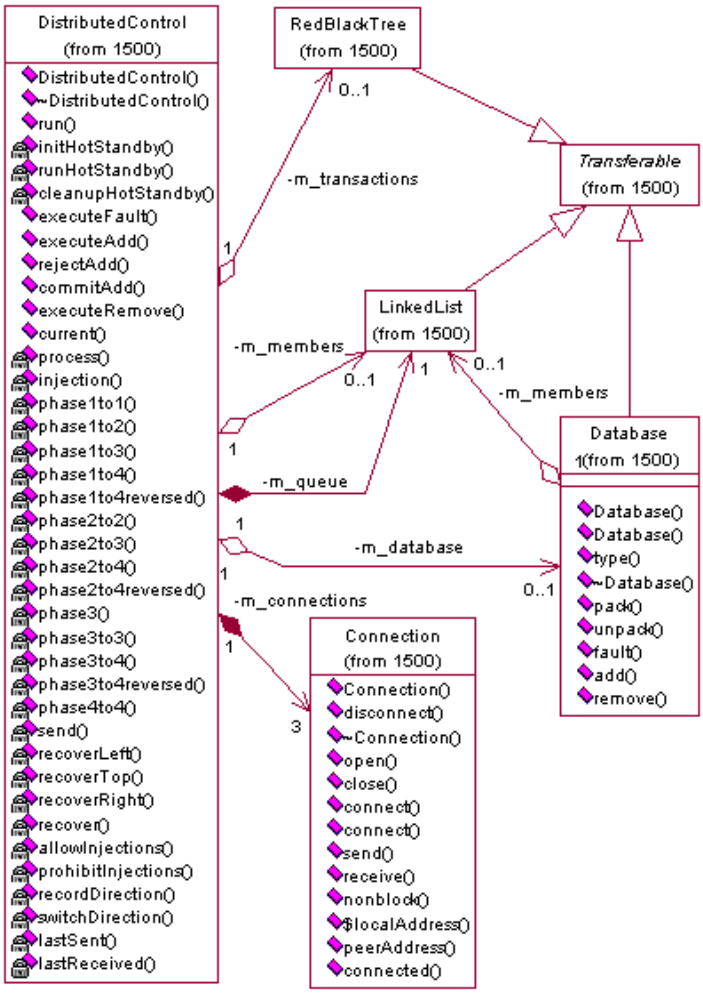


Figure 4.6: Distributed Control Class Diagram

The distributed control object provides an asynchronous communication behavior i.e. a transmission is processed and resulting transmissions are forwarded immediately after receiving. This is performed by one process only. The sending and receiving of transmissions is executed without any handshaking, so that a transmission is sent via the write operation of a network socket without a receive confirmation of the peer application. The TCP/IP protocol provides its own confirmation algorithm via a sliding window strategy. Transmissions are processed fairly from each direction i.e. from the left, the right or the client. A ring server also processes ring transmissions if no client is locally connected or injecting transmissions. A client can connect to the server, inject transaction requests and disconnect from the local server.

4.2 Implementation

The implementation of the prototype (7.2) in C++ is split into several files. Every class has a header file (ClassName.h) containing the class declaration and a source file (ClassName.cpp) containing the class implementation. Short and frequently accessed methods are implemented inline in the class declaration. Some class name declarations are defined additionally separately using forward declarations in a separate header file (Forwards.h) due to recursive 'has a' relationships (m to n). System wide constants are defined in the definition file (Defines.h) and system wide data types are defined in the type declaration file (Types.h). The system include files (System V) are included into the system includes file (System.h). Class header files include each other only if the class declaration it explicitly demands e.g. for member objects or method calls inside inline methods.

A class header file is included into a class source file only if the class declaration is needed by the implementation. The forward declaration file, the definition file, the type declaration file and the system includes file are included into the general header file (Includes.h) to provide a globally unique order of definitions and declarations to all class header files. The make file of the prototype provides two targets. One with debugging information for tests using a debugger and one without for performance tests. During the make process, the compiler builds the executable program using the linker directly from the class source files, where an extra source file (Main.cpp) contains the main function. A separate object file for each class is not created due to the complicated inter-class dependencies ('has a' and 'is a').

4.3 Tests

The tests are executed using the previously defined three different testing techniques. The dry run partially tests the distributed state-replication algorithm based on a test of the Atomic Broadcast in a bidirectional connected ring. The component test covers the check of major software components such as the balanced binary tree, the linked list, the object transfer mechanism and the non-blocking communication. The system test explores the distributed state-replication algorithm behavior. It is applied to every version of the implementation of the prototype if negative test results lead to changes and new versions. The preliminary system design is reviewed during the tests to discuss occurred problems and propose solutions for a complete review and analysis based on experimental results in the future, so that an evaluation of the prototype is performed.

4.3.1 Dry Run Test

The Transaction Control Algorithm with Distributed Agreement in an asynchronous bidirectional connected ring consists of an Atomic Broadcast of Transactions followed by a Distributed Agreement on the transaction execution result. The algorithm consists of six Reliable Broadcasts interleaved into four phases where transmissions are asynchronously forwarded in both directions in the server ring,

where the first Reliable Broadcast is blocked in its second phase to atomically broadcast all simultaneous transactions at once and to sequentialize the Atomic Broadcasts of all sequential transactions. The dry run test focuses on the Atomic Broadcast and only a part of Distributed Agreement i.e. on the phases one, two and three, since the correct behavior of the complete Distributed Agreement can be derived from the result.

Atomic Broadcast

The four phases of the Transaction Control Algorithm with Distributed Agreement in a bidirectional ring are cut down for the test to three phases, where the Atomic Broadcast is executed in the first two phases. This dry run test (fig. 7.1, 7.2, 7.3, 7.4, 7.5, on pages 64-68) shows that the Atomic Broadcast orders transactions correctly despite simultaneous or sequential injections. To simplify the state of a transaction at a member, H is introduced for the hold state, P is used for the pending state and D marks the second phase of the Distributed Agreement. The test of sequential injected transactions (fig. 7.2) shows that they have always a distance of at least one state per step at each member, so that the phases of the Atomic Broadcast and Distributed Agreement do not interfere each other. They are sequentialized for sequential injected transactions and the same for simultaneous transactions.

Transaction Control

Since transactions are ordered correctly and transmissions are sequentialized for sequential injected transactions and overlapped for simultaneous injected transactions, the four-phase Transaction Control Algorithm with Distributed Agreement in a bidirectional ring is correct. Furthermore, the Transaction Control Algorithm with Distributed Agreement is correct despite member faults if the member fault recovery send lost transmissions again in the same order as they were sent before and if already received transmissions are filtered. This dry run test result is also applicable to the Transaction Control Algorithm with Reliable Broadcast in a bidirectional ring and to the simple Transaction Control Algorithm, which is based on the Atomic Broadcast only, in a bidirectional ring, since the used Atomic Broadcast and the used Reliable Broadcast are tested.

4.3.2 Component Test

The prototype consists of four major components. The red-black tree provides a balanced binary tree, where transactions are stored using the transaction id as key. The linked list realizes a list with a wide variety of methods to store a list of members or a list of transaction execution results. The object transfer mechanism implements an object transport from one server to another, where an object of the same class with the same members is created at the other side. The non-blocking communication is used to enable the prototype to process incoming transmissions as soon they arrive and in a fair share. The component test of the prototype evaluates these single software components to decrease possible incorrect behavior in the system test of the prototype, since such problems are hard to detect and to resolve in the system test due to the system complexity.

Red-Black Tree

The red-black tree class implementation is tested by evaluating the correctness of all provided methods. The 'insert' method inserts an object and a key into a red-black tree object, where the balanced binary tree is maintained correctly. This is tested with increasing, decreasing and random key values with a large number (6000) of keys and objects to test also the log(n) operation counts. The red-black tree throws an exception if a key is inserted twice, because the key i.e. transaction id is unique and should not be inserted twice. The list iterator retrieves the link to the 'prev', 'next', 'last' or 'first' key and is tested after the keys and objects are inserted into the red-black tree. It retrieves a NULL

pointer if no previous, next, last or first key is available. The 'contains' method retrieves a true boolean value if the given key is in the tree.

Keys and objects can be retrieved using the 'key' or 'object' method with a given key which fulfills the equal operation to the key stored in the tree. The 'remove' method correctly removes an object from the tree if the given key fulfills the equal operation to the key stored in the tree, where the 'remove all' method removes all objects from the tree without needing a key. The 'free' method deletes the object additionally, where the 'free all' method removes and deletes all objects from the tree without needing a key. The test results show a correct behavior of the red-black tree class implementation. The balanced binary tree is always maintained correctly. The 'pack' and 'unpack' methods, i.e. the implementation of the abstract methods of the super class Transferable, are tested in the object transfer mechanism test.

Linked List

The linked list is tested similar to the red-black tree by evaluating the correctness of all provided methods, but with the insertion order instead of the key order. The 'add' method inserts an object before the current position of the list iterator and the 'append' method inserts an object after the current position of the list iterator. The list iterator points always to the inserted object after insertion. The insert method throws an exception if the list iterator do not point to any object and the list is not empty e.g. if a list end is reached by the 'next' or 'prev' method of the list iterator. This is tested with a large number (6000) of objects to check also the linear operation counts. The list iterator retrieves the link to the 'prev', 'next', 'last' or 'first' object and is tested after the objects are inserted. It retrieves a NULL pointer if no previous, next, last or first object is available.

The 'remove' method removes the current object to which the iterator points, where the 'remove all' method removes all objects from the list. The 'free' method deletes the object additionally, where the 'free all' method removes and deletes all objects from the list. The search method looks up an object in the list, retrieves a true boolean value and seeks the list iterator to the found object if the object is in the tree. It returns false otherwise. The test results show a correct behavior of the linked list class implementation. The insertion order is always maintained correctly. The 'pack' and 'unpack' methods, i.e. the implementation of the abstract methods of the super class Transferable, are tested in the object transfer mechanism test.

Object Transfer Mechanism

The object transfer mechanism is tested without communication by packing and unpacking objects such as a filled red-black tree or linked list into and from a byte oriented buffer. The test results show a correct behavior, where objects of the same class with the same members are created. The object transfer mechanism throws an exception if an unknown class type occurs. But, the object transfer mechanism works without checksums or signatures, so that data corruption may lead to unpredictable behavior e.g. to memory access violation. All transferable objects are transfered correctly, where the order of a linked list is maintained as well as the order of a red-black tree. The public static helper methods of the Transferable class pack and unpack all data types correctly, where unique data types with defined bit length and byte order are maintained.

Non-Blocking Communication

The non-blocking communication is tested by applying the object transfer mechanism to the non-blocking socket communication, where objects of the same class with the same members are created after the transfer. The sender process is killed during sending to test the correct behavior in case of communication faults, where the object is not unpacked and a communication fault is reported. The sending and receiving of the buffer is executed in single steps using the debugger at the sender and

receiver to test the blocking communication until all data is sent and received. The test results show a correct behavior of the non-blocking communication implementation.

4.3.3 System Test

The system test of the prototype evaluates the correctness of the implemented state-replication algorithm with one, two or more than three servers. It checks the ring synchronization, i.e. the phase transitions which need to be executed correctly despite congestion in one direction. The transaction ordering is verified to ensure a globally unique and semantically correct order of transactions, where simultaneous transactions are shared fairly and sequential transactions are sequentialized. The correctness of the member fault recovery algorithm is checked as well as the member admission and the member removal transaction execution. The linear scalability and the heterogeneity of the implementation are also verified. All these tests are performed using the implementation of the prototype, where a version change requires to execute all tests again.

Ring Synchronization

The Transaction Control Algorithm with Distributed Agreement in an asynchronous bidirectional connected ring consists of an Atomic Broadcast of transactions followed by a Distributed Agreement on the transaction execution result. The algorithm consists of six Reliable Broadcasts which are interleaved into four phases, where transmissions are asynchronously forwarded in both directions in the server ring. A phase transition is executed if two members recognize that transmissions of the same id of the same phase type cross. Only the injection, phase transitions and the fair share synchronizes these transmissions. Since the communication in both directions is independent and mostly asynchronous, a transmission of one direction may perform one round more than the transmission of the other direction.

So that a phase transition is executed if two members recognize that transmissions of the same id cross, where the phase transition type depends on the types of crossing transmissions. But, the same phase transitions must executed always at two neighbor members in the ring, so that a transmission which is received too late performs a phase transition related to the previous received transmission from the other side. Tests of such phase transitions in a two, or more than three server ring show that the ring works correctly if one, two or three clients inject transaction requests. But the transaction order is different despite correct transaction order (transaction id) if more than three clients inject transaction requests. A subset of servers commits all transactions in the same and right order and the other servers commit some transactions in the same but wrong order.

More precisely, some transactions (0.001%) are committed earlier as they should at a subset of the server group. This negative test result leads to the conclusion that either the phase transitions are not executed correctly or the Atomic Broadcast lacks of correctness. Both may produce such results, where transaction control states and transmission phases are not sequentialized correctly. A complete review and analysis based on experimental results in the future should perform dry run tests with such a test environment. The main goal of a review and analysis should be a general correctness proof of phase transitions in an asynchronous bidirectional connected ring and a correctness proof of the Atomic Broadcast in an asynchronous bidirectional connected ring based on the previous correctness proof. Both proofs should cover simultaneous and sequentialized transaction requests as well.

Transaction Ordering

The transaction order is *almost* correct. Only the previous described problem exists, where some transactions are committed earlier as they should. Excluding this problem, transactions are ordered, executed and committed correctly. Sequential transactions have increasing transaction numbers and

simultaneous transactions have always the same transaction number, where the transaction id consists of transaction number and member id. The semantically correct order is maintained as well as the global order. The transaction order shows always a fair share, so that simultaneous transactions of all members which are executed and committed in one block occur if members inject transactions simultaneously. So that starvation of members caused by transactions injected by a subset of members which overload the system do not occur in any test configuration.

Fault Recovery

Single member faults, multiple member faults and cascaded member faults are detected correctly by the neighbors. Phase two and three transmissions are sent again in the right order, but phase one transmissions are incorrectly sent in increasing order. This is caused by an incorrect implementation of the previously defined recovery algorithm, where all transmissions, except phase four transmissions are sent again in the same order as previously sent. Phase one transmissions cannot be sent in increasing order on both sides and cannot be sent in increasing order on one side and in decreasing order on the other side. A correct recovery needs to additionally monitor the sending order of transmissions, because transmissions are sent independent from their transaction order. The current version of the prototype do not monitor the sending order, so that a next version should fix that problem.

Previously received transmissions are correctly filtered by the member on the other side. The 'member fault' transaction is injected by only one member at the end of the fault recovery, but causes after its execution or commit a system halt. Since other simultaneous transactions are executed and committed after the recovery until the system halt, the previously mentioned transmission order problem may be the reason. Currently a server is allowed to inject only one transaction at a time, so that a server waits until its transaction is executed, committed and deleted to inject the next transaction. If the Transaction Control algorithm is not correctly executed, a server may wait forever to finish the Transaction Control algorithm for its transaction. The current version of the prototype do not inject 'member fault' transactions any more for other test purposes.

The recovery algorithm first tries to reconnect to the lost member to ensure that network overload was not the reason for the TCP/IP error. Every try of the recovery algorithm to connect a member consists of several tries with increasing delay and a timeout to give the other side the chance to start its own recovery algorithm. So that members try to connect to each other in a time window which is produced by the timeouts of both sides. However, the recovery algorithm must ensure that a member has still its current state after a reconnect. A server process which is killed and restarted during the time window may accept the recovery. Future versions of the prototype should provide two different protocols, which are based on handshaking and bidirectional information exchange, to differentiate between a member admission and member fault recovery on directly involved members.

Member Admission

Tests with 'member admission' transactions show that not only the member removal needs the member fault recovery. Since the ring communication is asynchronous and bidirectional, the dedicated previous and the dedicated next member execute a 'member admission' transaction. Assumptions which one executes it at first cannot be made, so that the disconnect is always unexpected. Additionally, the member fault recovery is already executed if the new member cannot be connected or fails during the transaction execution. The current version of the prototype uses the member fault recovery to execute member admissions. The dedicated previous member adds the new member to the members list and disconnects its server socket. The member fault recovery connects the new member or the dedicated next member if it fails.

The dedicated next member disconnects its client socket and connects the new member and falls into the member fault recovery if it fails. This partially synchronizes dedicated previous and the

dedicated next member, but a deadlock or incorrect member list entries may occur. Additionally, the incorrect member fault recovery algorithm causes a system halt after a failed member admission. The implementation do not guarantee that a member admission can be executed by either the dedicated previous or the dedicated next member and is ignored if it is already executed. Future versions should guarantee that.

Member Removal

The member removal is executed by a self-disconnect of the member to be removed and the member fault recovery of the previous and next member. The same problems as in the member admission occur, since the removal is asynchronous and at least one member do not expect the removal. Additionally, the incorrect member fault recovery algorithm causes a system halt after the member removal. The current version of the prototype uses the member fault recovery to execute member removals. The test results and review of the fault recovery algorithm shows that a member removal needs to be controlled by one neighbor member of the member to be removed. The implementation do not guarantee that a member removal can be executed by either the previous or the next member and is ignored if it is already executed. Future versions should guarantee that.

Linear Scalability and Heterogeneity

The system shows a linear scalable behavior despite incorrect algorithms. The transaction injection-to-commit delay time depends linear on the number of members and linear on the number of simultaneous transactions, which may be different. But the transaction injection-to-commit delay depends also on the system performance and on the implementation efficiency. Dynamic memory allocation and method resolution of object-oriented languages should be avoided in the future. Exact and extensive measurements are not executed, since the incorrectness of the system do not justify such tests. The heterogeneity is provided by the use of System V socket communication, TCP/IP single-cast and ANSI-C/C++. But future versions should provide standard TCP/IP port numbers and may piggyback the communication on a standard protocol, such as HTTP.

5 Conclusion

This work aimed to develop a distributed control for the Harness distributed virtual machine that avoids single point (or set of points) of failure for a distributed heterogeneous system architecture. A system analysis of distributed system architectures currently used in heterogeneous distributed computing for parallel and scientific applications has shown that such systems are asynchronous but may use partially synchronous communication to detect and to distinguish different classes of faults and failures. Furthermore, a large scale distributed system with a huge number of servers residing on different geographical locations needs to be split in several different groups with hot-standby or warm-standby high availability to achieve feasibility and efficiency. This provides also an adjustable level of fault-tolerance at each geographical location.

The aspired linear scalability have lead to the development of linear-scalable distributed algorithms in a linear-scalable peer-to-peer network architecture. Asynchronous group communication services, such as Reliable Broadcast, Atomic Broadcast, Distributed Agreement, Transaction Control and Membership, were analyzed to develop linear-scalable group communication services for an unidirectional and a bidirectional connected asynchronous peer-to-peer ring architecture. The Transaction Control group communication service was newly introduced as state-replication service to distinguish the transaction processing of different types of distributed systems, where active transactions execute state changes using replicated data or non-replicated data of one or more servers and inactive transactions report state changes using replicated data only.

The introduced Transaction Control algorithms are applicable for passive fault-tolerant distributed databases as well as for active fault-tolerant distributed control mechanisms. Additionally, all algorithms are linear scalable due to the linear scalable peer-to-peer ring architecture. In contrast to previous algorithms, no control token [20, 21, 22] or time stamp [18, 19, 20, 21, 22] is used. So that the developed group communication services provide equal server responsibilities i.e. symmetric algorithms and are independent from the system time. Furthermore, a globally unique and semantically correct transaction order is achieved without blocking the transfer of unreliable broadcasted information [6, 9, 10] and without optimistically executing and undo transactions [25, 26], where the member fault recovery involves only two members of a server group.

However, only a prototype which implements the most complicated Transaction Control algorithm was realized due to the complexity of the distributed system and the early development stage of the introduced algorithms. The main goal, which was the development, implementation and test of a distributed state-replication algorithm for a consistent fault-tolerant distributed control in a distributed heterogeneous system architecture which meets the criteria of correctness, fault-tolerance, scalability, heterogeneity and efficiency, was only partially reached. The test results have shown a partially incorrect behavior of the Transaction Control algorithm, the member fault recovery algorithm, the member admission and removal transaction execution. Further research should perform a complete review and system analysis based on the experimental results.

Despite substantial research effort, the design and implementation of high-available heterogeneous distributed system software is still a complex and intriguing endeavor. Distributed algorithms for consistent replication of information or for detection of and recovery from faults and failures in a heterogeneous distributed system architecture highly depend on the distributed system model. The main achievement of this work was the development of a distributed control for the Harness

distributed virtual machine based on an analysis of the distributed system model. Group communication algorithms in ring architectures are not new in the research area of fault-tolerant distributed computing, but the conditions and assumptions of previous work do not match the requirements of heterogeneous distributed computing.

The implemented prototype was also a success despite negative experimental results. The experiments have shown that some problems are solved, e.g. fair share, total order and linear scalability, and some problems still exists, e.g. phase transitions and member fault recovery, and need to be solved. Future work should analyze the test results using a complete review of the system design. The next prototype or a final implementation requires a more precise specification of algorithms and data structures to improve the correctness and the performance of group communication services. The next development stage should target to a set of C functions with a correct implementation of the Transaction Control group communication service with and without Distributed Agreement, since both are needed in the distributed control.

6 References

[1] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam; *PVM: Parallel Virtual Machine; A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994

[2] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra; *MPI: The Complete Reference;* MIT Press, Cambridge, MA, 1996

[3] G.A. Geist, J.A. Kohl, P.M. Papadopoulos; *PVM and MPI: a Comparison of Features;* May 1996

[4] G.A. Geist, J.A. Kohl, P.M. Papadopoulos, S.L. Scott; *Beyond PVM 3.4: What We've Learned, What's Next, and Why;* Oak Ridge National Laboratory, Computer Science and Mathematics Division, USA

[5] J. Dongarra, A. Geist, J.A. Kohl, P.M. Papadopoulos, V. Sunderam; *HARNESS: Heterogenous Adaptable Reconfigurable NEtworked SystemS;* Oak Ridge National Laboratory, Computer Science and Mathematics Division, USA

[6] G.A. Geist, J.A. Kohl, S.L. Scott, P.M. Papadopoulos; *HARNESS: Adaptable Virtual Machine Environment For Heterogeneous Clusters;* Parallel Processing Letters, Vol. 9, No. 2, (1999), pp 253-273

[7] M. Migliardi, J. Dongarra, G.A. Geist, V. Sunderam; *Dynamic Reconfiguration and Virtual Machine Management in the Harness Metacomputing System;* Emory University, Department of Mathematics and Computer Science, Atlanta, USA

[8] M. Migliardi, V. Sunderam; *PVM Emulation in the Harness Metacomputing System: a Plug-in Based Approach;* Emory University, Department of Mathematics and Computer Science, Atlanta, USA

[9] G.A. Geist, J. Dongarra, V. Sunderam; *HARNESS: Exploring the Frontiers of Heterogeneous Distributed Computing (Presentation);* Oak Ridge National Laboratory, Computer Science and Mathematics Division, USA

[10] S.L. Scott; *Distributed Control in the Harness Virtual Machine (Presentation);*, Oak Ridge National Laboratory, Computer Science and Mathematics Division, USA

[11] F. Cristian; *Synchronous and Asynchronous Group Communication;* Communications of the ACM, Vol. 39, No. 4, April 1996, pp 88-97

[12] F. Cristian; *Synchronous and Asynchronous Group Communication (Long Version);* Department of Computer Science and Engineering, University of California, San Diego, USA

[13] S. Toueg, K.J. Perry, T.K. Srikanth; *Fast Distributed Agreement (Preliminary Version);* Department of Computer Science, Cornell University, Ithaca, New York 14853, USA,1985, ACM Digital Library

[14] G. Neiger, S. Toueg; *Automatically Increasing the Fault-Tolerance of Distributed Systems (Preliminary Version);* Department of Computer Science, Cornell University, Ithaca, New York 14853, USA, May 1988, ACM Digital Library

[15] T.D. Chandra, S. Toueg; *Unreliable Failure Detectors for Asynchronous Systems (Preliminary Version);* Department of Computer Science, Cornell University, Ithaca, New York 14853, USA, 1991, ACM Digital Library

[16] T.D. Chandra, S. Toueg; *Unreliable Failure Detectors for Reliable Distributed Systems;* I.B.M Thomas J. Watson Research Center, Hawthorne, New York and Department of Computer Science, Cornell University, Ithaca, New York 14853, USA, 1991, ACM Digital Library

[17] D.D. Kandlur, K.G. Shin; *Reliable Broadcast Algorithms for HARTS;* The University of Michigan, ACM Transactions on Computer Systems, Vol. 9, No. 4, November 1991, pp 374-398

[18] I. Keidar, D. Dolev; *Efficient Message Ordering in Dynamic Networks;* Computer Science Institute, The Hebrew University of Jerusalem, 1996, ACM Digital Library

[19] D. Dolev, D. Malki; *The Transis Approach to High Availability Cluster Communication;* Communications of the ACM, Vol. 39, No. 4, April 1996, pp 64-70

[20] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal and P. Ciarfella; *The Totem single-ring ordering and membership protocol;* ACM Transactions on Computer Systems 13, 4 (Nov. 1995), pp 311 - 342

[21] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos; *Totem: a fault-tolerant multicast group communication system;* Communications of the ACM 39, 4 (Apr. 1996), pp 54 - 63

[22] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith and R. K. Budhia; *The Totem multiple-ring ordering and topology maintenance protocol;* ACM Transactions on Computer Systems 16, 2 (May. 1998), pp 93 - 132

[23] F.C. Gärtner; *Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments;* ACM Computing Surveys, Vol. 31, No. 1, March 1999

[24] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso; *Understanding Replication in Databases and Distributed Systems;* Swiss Federal Institute of Technology (EPFL), Operation Systems Laboratory, Lausanne and Swiss Federal Institute of Technology (ETHZ), Institute of Information Systems, Zürich, ACM Digital Library

[25] B. Kemme, F. Pedone, G. Alonso, A. Schiper; *Using Optimistic Atomic Broadcast in Transaction Processing Systems;* Swiss Federal Institute of Technology (ETHZ), Institute of Information Systems, Zürich, ACM Digital Library

[26] M. Patino-Martinez, R. Jiminez-Peris, B. Kemme, G. Alonso; *Scalable Replication in Database Clusters;* Technical University of Madrid, Facultad de Informatica, Boadilla del Monte, Madrid, Spain and Swiss Federal Institute of Technology (ETHZ), Department of Computer Science, Zürich, ACM Digital Library

[27] R. I. Resnick; *A Modern Taxonomy of High Availability;* 1996, `http://www.interlog.com/~resnick/ron.html`

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest; *Introduction to Algorithms (MIT Electrical Engineering and Computer Science);* MIT Press; Cambridge, MA, 1990

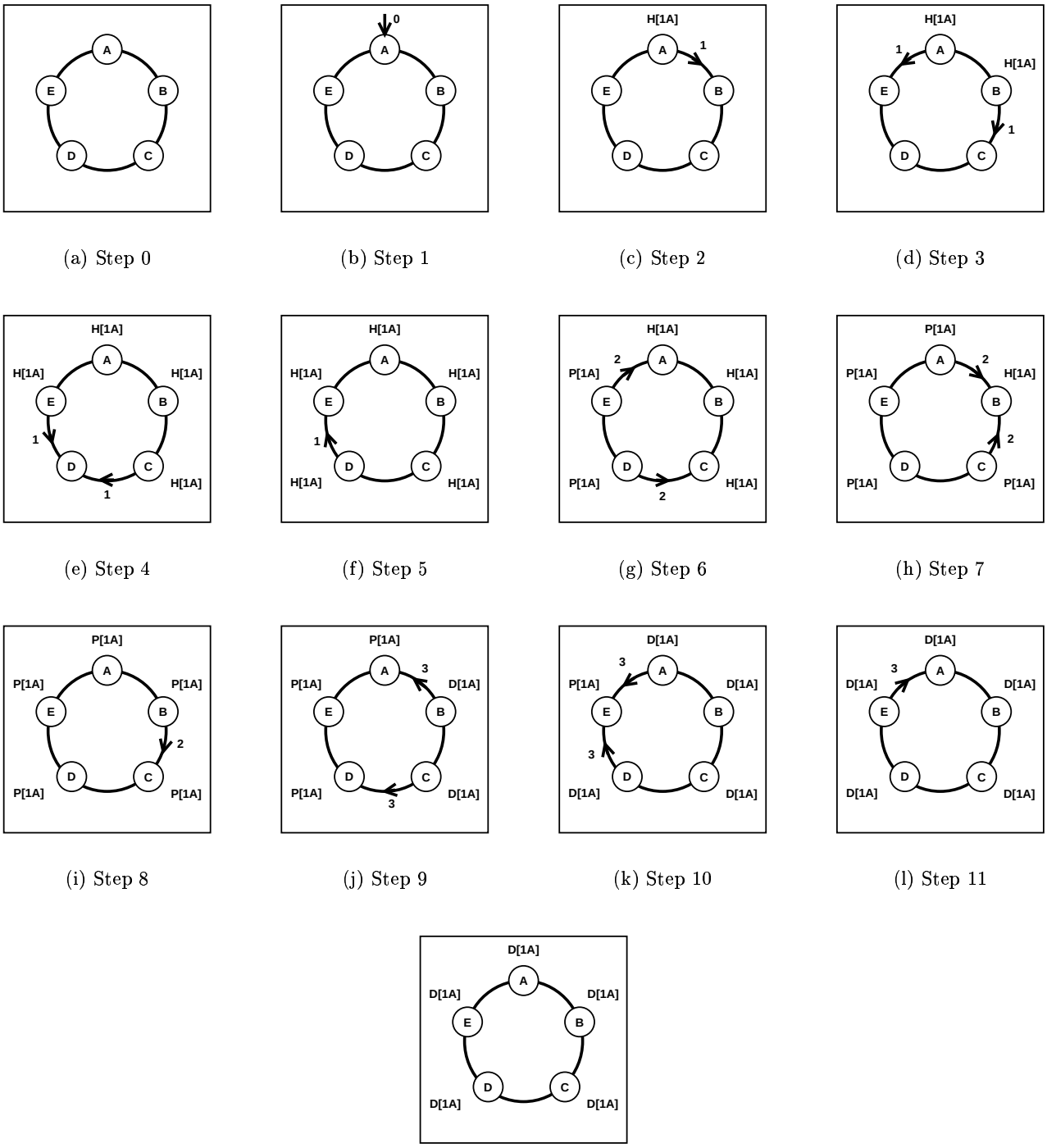
[29] Object Management Group; *OMG Unified Modeling Language Specification*; Version 1.3, June 1999

[30] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch; Design Patterns; Addison-Wesley Pub Co, 1995

[31] W. Richard Stevens; Advanced Programming in the UNIX Environment; Addison-Wesley Pub Co; 1992

7 Appendices

7.1 Dry Run Test



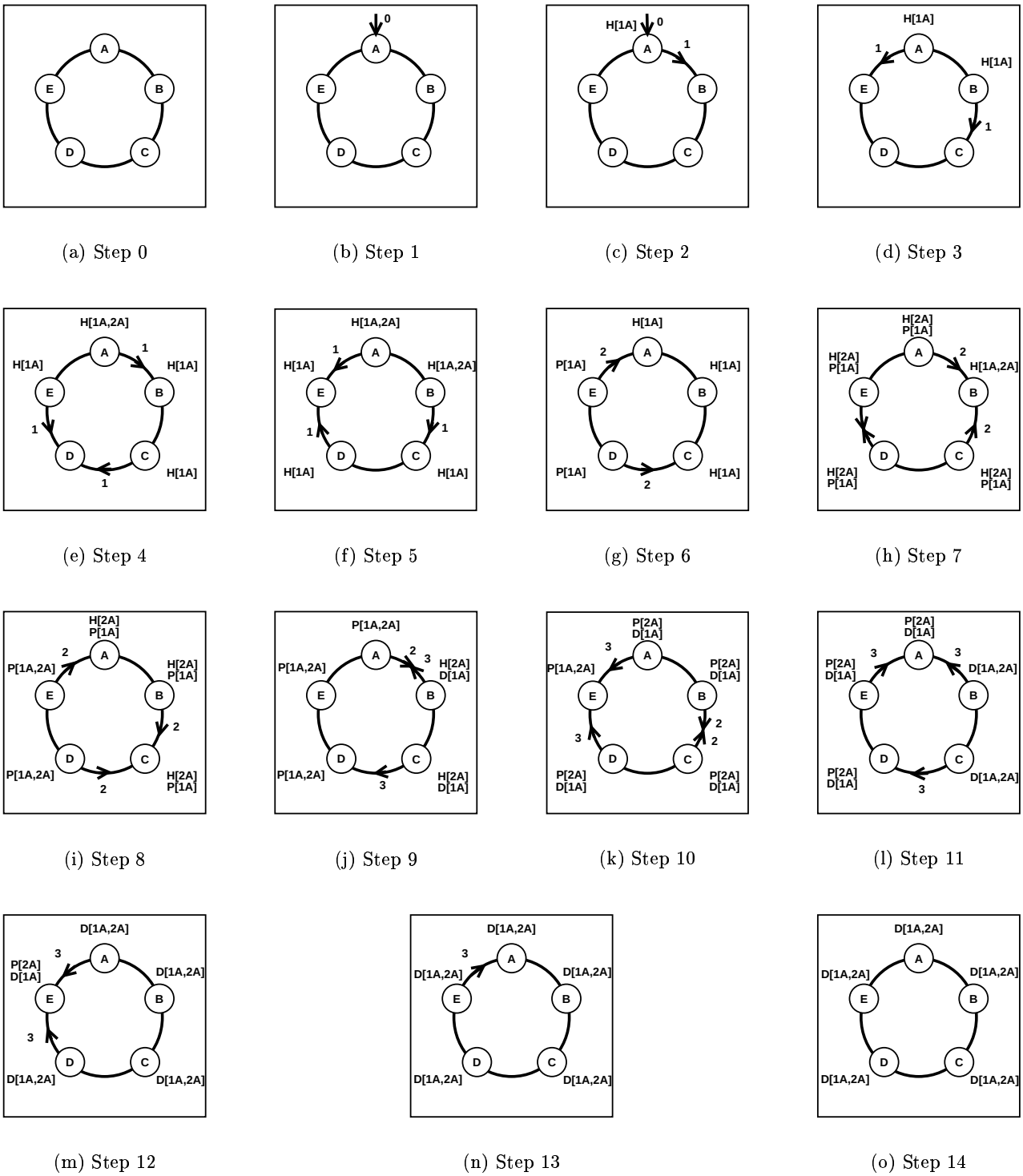
(a) Step 0 (b) Step 1 (c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5 (g) Step 6 (h) Step 7

(i) Step 8 (j) Step 9 (k) Step 10 (l) Step 11

(m) Step 12

Figure 7.1: Atomic Broadcast of one Transaction



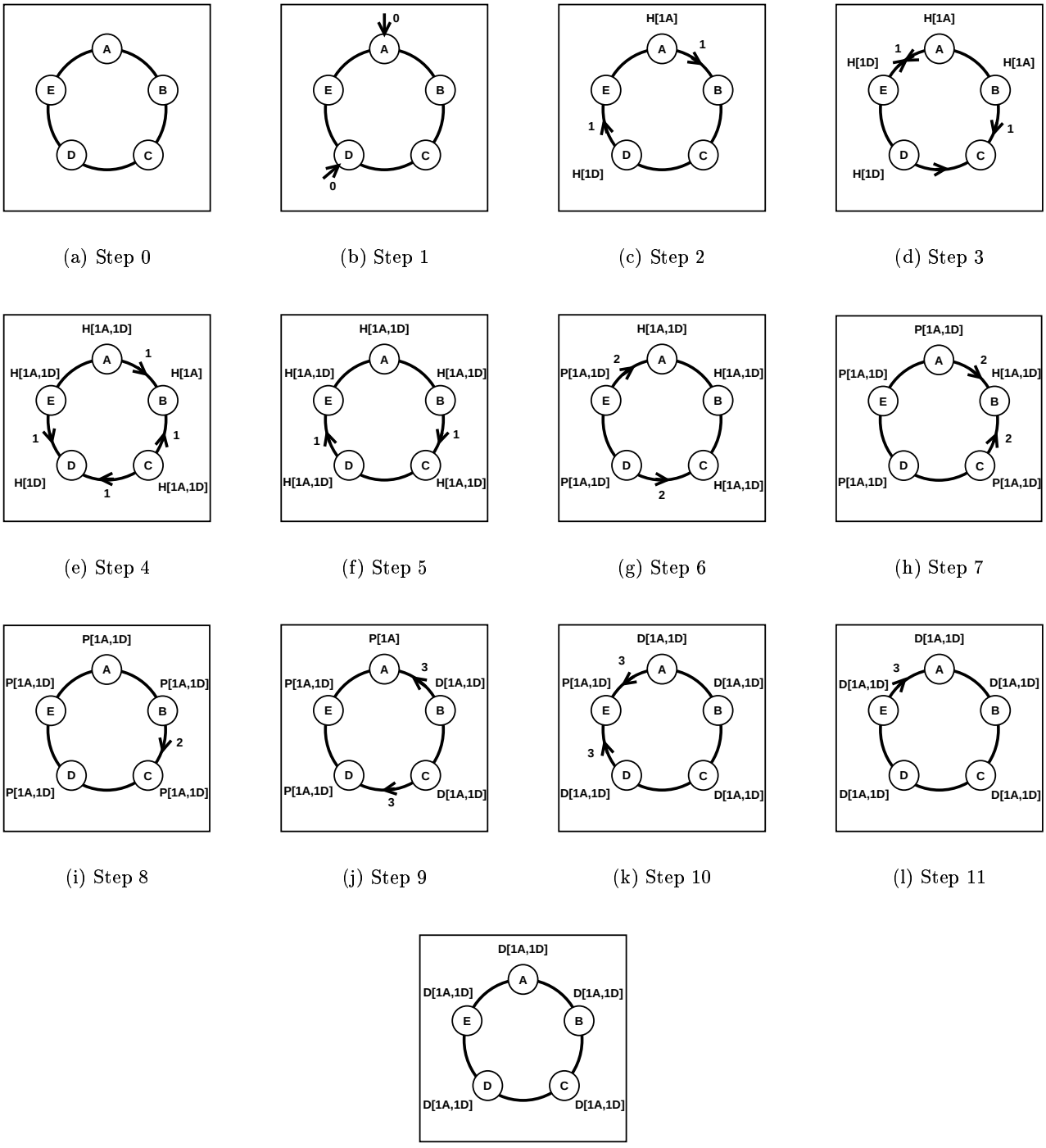
(a) Step 0 (b) Step 1 (c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5 (g) Step 6 (h) Step 7

(i) Step 8 (j) Step 9 (k) Step 10 (l) Step 11

(m) Step 12 (n) Step 13 (o) Step 14

Figure 7.2: Atomic Broadcast of two Transactions



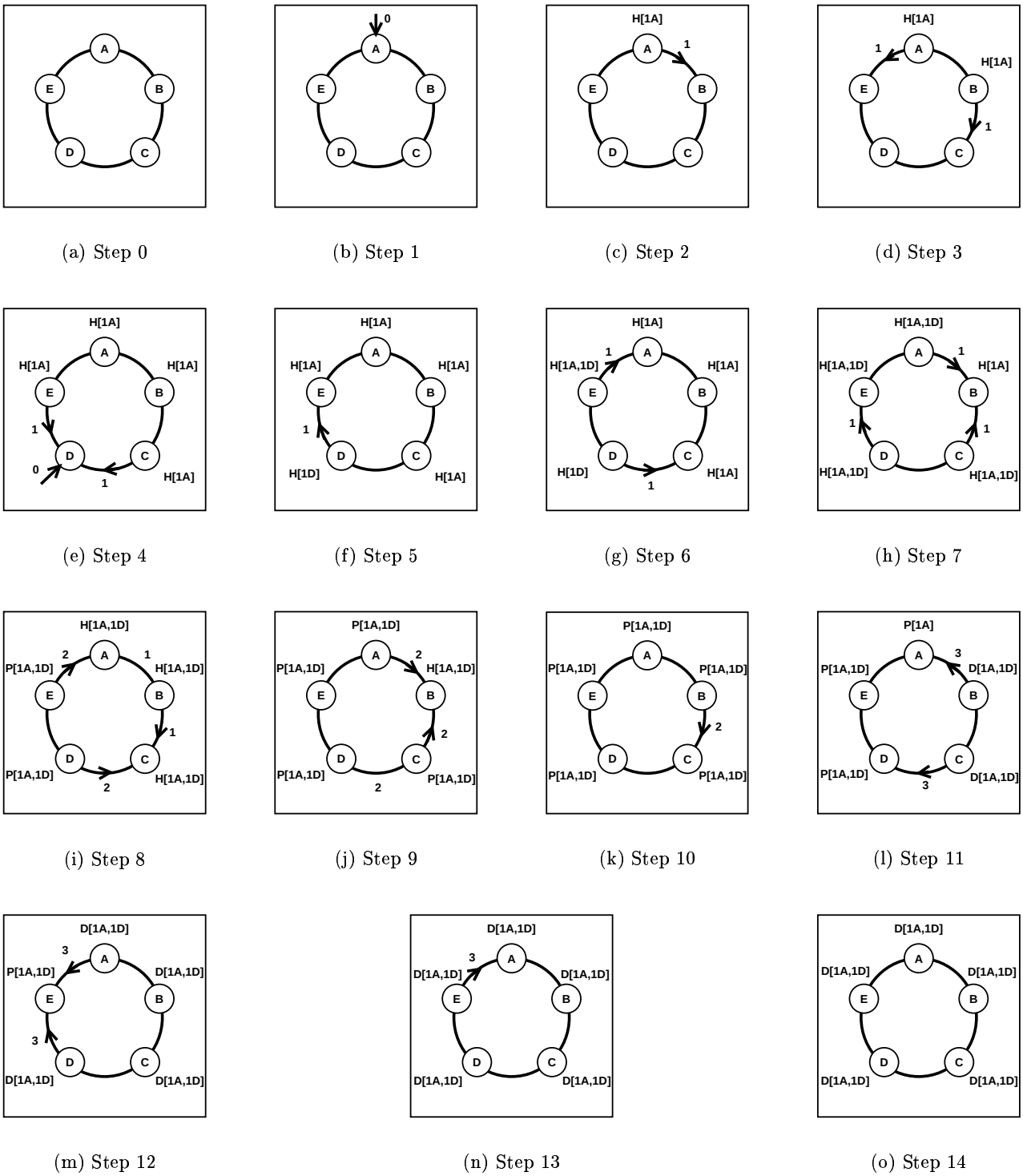
(a) Step 0 (b) Step 1 (c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5 (g) Step 6 (h) Step 7

(i) Step 8 (j) Step 9 (k) Step 10 (l) Step 11

(m) Step 12

Figure 7.3: Atomic Broadcast of two simultaneous Transactions (Case 1)



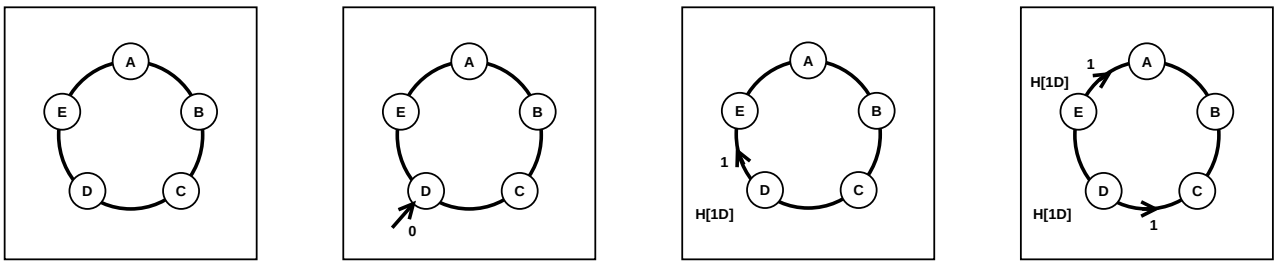
(a) Step 0 (b) Step 1 (c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5 (g) Step 6 (h) Step 7

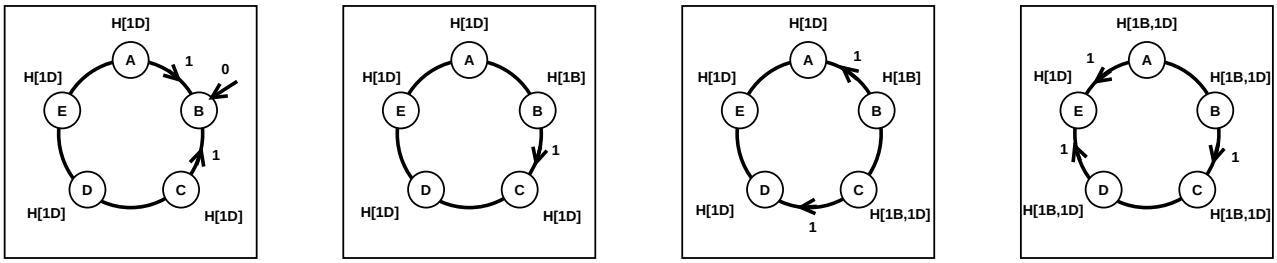
(i) Step 8 (j) Step 9 (k) Step 10 (l) Step 11

(m) Step 12 (n) Step 13 (o) Step 14

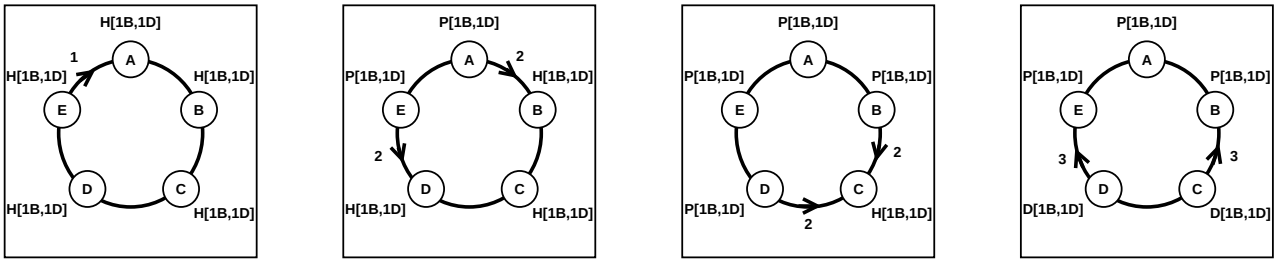
Figure 7.4: Atomic Broadcast of two simultaneous Transactions (Case 2)



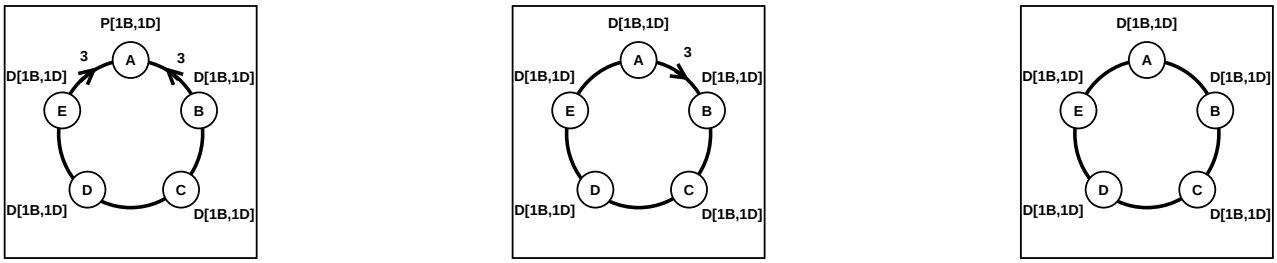
(a) Step 0 (b) Step 1 (c) Step 2 (d) Step 3



(e) Step 4 (f) Step 5 (g) Step 6 (h) Step 7



(i) Step 8 (j) Step 9 (k) Step 10 (l) Step 11



(m) Step 12 (n) Step 13 (o) Step 14

Figure 7.5: Atomic Broadcast of two simultaneous Transactions (Case 3)

7.2 Program Listing