A Virtual Shared Disk on Distributed Redundant Storage

Stefan Vijzelaar

February 9, 2010

Abstract

Simulating a shared disk on distributed redundant storage requires a consistent ordering of disk operations. Consensus on such an ordering is impossible in asynchronous systems with failing processes. Consistency can however be achieved without reaching consensus, even when only a majority of processes is available. This paper shows how to implement a virtual shared disk, using an appropriate consistency model. Algorithms are verified using the Spin model checker.

1 Introduction

Storage sharing is an important application of current network technology, ranging from sharing files between computers in a home network to consolidating storage in an enterprise server environment. These applications can choose from a variety of hardware and software solutions. Storage can for example be shared at block level: using host bus adapters attached to shared disks, or network adapters connected to a storage area network. Storage can also be shared at file level: using distributed file systems with storage servers, or network file systems on network attached storage. Storage systems can even be used to complement each other: shared disk file systems can transform shared block storage into shared file storage, storage servers of a distributed file system can use shared disks as backing storage. There are a lot of options to stack, mix and match storage solutions.

Reliability is a big concern for storage systems, especially when storage is shared. Sharing storage can increase the impact of data loss, should the storage system fail. The solution is to introduce redundancy into the system, allowing for a certain amount of backing storage to fail without jeopardizing integrity. Redundant arrays of inexpensive disk (RAID) are a prime example: data is stored on multiple disks and can survive disk failures. However, RAID and similar techniques are designed to run on a single machine with directly attached storage. Even when shared block storage is used, a RAID implementation can only run on a single node at a time. It is not meant to run concurrently. This still leaves the node as a single point of failure.

A single point of failure can be removed by either using fail-over or making it redundant. Fail-over implementations let another node take over when a node fails. The problem then shifts to detecting node failure: in an asynchronous system it is impossible to distinguish between a slow and a failed node. Hardware solutions exist to ensure that a presumably failed node stays down, for example by cutting its power, but these solutions add complexity to the system. Redundant implementations make sure no single point of failure exists. The problem then becomes one of coordination between participating nodes, which might prove difficult since deterministic consensus in an asynchronous system is impossible when nodes can fail. Hardware solutions exist in the form of redundantly connected RAID appliances, but these are generally expensive. A more cost effective solution would be a redundant implementation in software.

This paper presents a virtual shared disk: a shared disk implementation in software, based on redundant distributed storage. Data is distributed over multiple nodes and can be accessed concurrently. Despite this, the system still behaves as a single shared disk. Shared disk semantics are ensured through a consistency model: read and write operations behave as if subject to some global order. For example, when a read or write finishes, no subsequent read can return an older value than the one just read or written. It is not necessary to have complete consensus on the global order. It is enough to prove that, for each execution, one or more possible orderings exist which are consistent with the observed read and write behavior. The consistency model used in this paper is called strict linearizability. It allows virtual shared disks to function as a drop-in replacement for shared disk hardware.

Messages sent concurrently by the protocol can arrive in different orders at their destinations; caused by arbitrary message delays introduced by the network. Correctness needs to be evaluated for each possible ordering of messages to ensure reliability and sequential consistency. Model checking is used to make sure the virtual shared disk behaves correctly. A Promela model of the algorithm is created and subsequently verified using the Spin model checker. The model checker builds a state space of the protocol and verifies its correctness.

The virtual shared disk is meant to run on plain commodity hardware: personal computers, some with local storage, connected through a network. Machines running the MINIX operating system and connected using an Ethernet network are used as a platform for an initial implementation. Since the solution is based on software, it does not require hardware shared disks, which can be especially expensive when redundancy is required. Together with other, for example open source, storage solutions, this gives access to an affordable redundant storage system.

This paper is structured as follows. Section 2 is an introduction to concepts of storage systems. It provides a basic overview of terminology and describes solutions which can be used to complement a virtual shared disk. Section 3 presents use cases for a virtual shared disk implementation: it can replace and improve upon the uses of conventional shared disks. Section 4 explores the theory available to construct virtual shared disks and investigates competing solutions.

A preliminary algorithm for virtual shared disks is introduced in section 5. It is followed by sections 6 and 7 respectively, which show the model checking and implemention of the algorithm. The preliminary algorithm can cope with asynchronous networks, but not failing processes. Section 8 presents a better algorithm to implement a virtual shared disk. In this algorithm, processes can fail as long as a majority remains responsive. Section 9 shows how model checking and implementation are combined by sharing C-code.

2 Storage system concepts

This section will provide a basic overview of concepts and terminology used in storage systems. It references solutions which can be used complementary to a virtual shared disk implementation. One of the most ubiquitous technology in storage systems, namely RAID, is discussed in section 2.1. It increases the reliability of directly attached storage. Section 2.2 shows how LVM can be used to divide storage in smaller pieces; resulting in flexible backing storage for one or more virtual shared disks. Dedicating a network to shared disks results in a SAN as presented in section 2.3. An alternative called NAS is shown in section 2.4, where files are shared over the network instead of block devices. Because it is typical to store some type of filesystem on a storage device, we look at shared disk filesystems in section 2.5 and distributed filesystems in section 2.6.

2.1 RAID

A redundant array of independent disks (RAID) combines multiple disks into a single virtual aggregate disk. The aggregate RAID disk can exploit the underlying parallelism to increase performance and reliability. The probability of a single disk failure however increases when combining multiple disks. RAID disks need to cope with disk failures, which can potentially destroy data stored on the aggregate disk. RAID uses mirroring or parity-based schemes to redundantly store data, ensuring data can survive partial failures of the array.

Different RAID levels exist, corresponding to the techniques used for data storage in the array. The more common levels are:

RAID 0 distributes data over the array by striping: data is split into fixed size stripes which are distributed in a round-robin fashion over the disks in the array. RAID 0 can however not handle disk failures, making it the only level without added redundancy. There are benefits though: thanks to the round-robin distribution of stripes, both read and write operations can be parallelized to increase performance; and since RAID 0 looses no storage to redundancy, each disk is fully utilized to increase storage of the aggregate disk. RAID 0 is used primarily to improve performance.

RAID 1 uses mirroring to store identical copies of data on two or more disks: the content of the aggregate disk is duplicated to all underlying disks. Redundancy is improved by each disk added to the array: an array with n disks can sustain $n-1$ failures, without losing data. Read performance can be increased by reading in parallel, although write performance is no faster than a single disk. No storage is gained when adding disks to the array. RAID 1 is used for redundancy, especially in two disk arrays, or when recovery from multiple failures is required.

RAID 5 uses parity stripes in addition to the data stripes of RAID 0. Parity stripes are calculated from groups of data stripes to gain redundancy. An array of n disks will contain 1 parity stripe for each group of $n-1$ data stripes. Stripes are laid out in the array such that no disk contains more than one data or parity stripe from the same group. On a single disk failure, a missing data stripe can

then be recalculated using the parity stripe. RAID 5 can not recover from more than one disk failure. By distributing the parity stripes over different disks, performance increases with the number of disks; although write performance will suffer from the need to calculate parity stripes. The array also effectively looses one disk worth of storage due to redundancy. RAID 5 is a compromise between redundancy and storage size, generally used on three or more disks.

RAID 6 uses two parity stripes per group. This level is very similar to RAID 5, but allows simultaneous failure of two disks instead of one. Especially when using a large number of disks, the chance of a simultaneous disk failure can become large enough to warrant this additional redundancy. Write performance can be lower than RAID 5, due to the more complex calculation of the two parity stripes. Two disks worth of storage are lost due to redundancy. RAID 6 is used when RAID 5 is not deemed secure enough, usually with larger numbers of disks.

RAID levels can be stacked to achieve an appropriate balance between performance, redundancy and storage. A common example is RAID 10: disks are first mirrored into aggregate disks for redundancy using RAID 1, after which these aggregate disks are striped together for performance using RAID 0. Less common is RAID 01, which is similar but opposite to RAID 10: instead of striping mirrored disks, it mirrors striped disks. RAID 10 is preferred, because recovery after a single disk failure requires more data to be resynchronized for RAID 01, due to it striping before mirroring. Another example is RAID 50, which stripes RAID 5 arrays together to increase performance. This stacking of storage techniques is a common practice for storage systems and not limited to RAID levels.

Mentioned before, is the increased chance of disk failure when an aggregate disk consists of a larger number of disks. One has to keep in mind that while multiple disk failures are less probable than single disk failures, single disk failures are more probable for an aggregate disk than a stand-alone disk. Disks have to be replaced quickly before the next failure occurs, which could result in data loss. Failing to do so in a timely manner only increases the chance of additional failures. RAID gives the user a window of opportunity to restore redundancy.

Although RAID prevents data loss from disk failure, transient or partial failures can still corrupt data. When disks become flaky or have bad sectors, it is up to the user to determine and replace the faulty drive to repair the array. Parity stripes stored in a bad sector may go unnoticed until a regular disk failure occurs, resulting in data loss. A similar situation can occur when power is lost while writing: parity stripes become inconsistent with data stripes. This last problem is called the write hole and affects both RAID 5 and RAID 6 arrays. Battery backed controllers and power supplies are used to reduce the risk. RAID on its own is only designed to handle complete disk failures.

2.2 Logical volume management

Logical volume managers are a more flexible alternative to partitions: they divide and combine available storage into virtual disks. These virtual disks can also be easily moved or re-sized. Examples are the Logical Volume Manager (LVM) for Linux, the Solaris Volume Manager (SVM), the Logical Disk Manager of Microsoft Windows, and the cross platform Veritas Volume Manager (VVM). Often RAID techniques are employed to ensure reliability; either by the volume manager itself, or the underlying virtual disks.

The mapping of storage into virtual disks is divided into steps. First the underlying disks, called physical volumes (PVs), are divided into fixed size chunks, called physical extents (PEs). These PEs, possibly originating from different physical volumes, are assigned to a volume group (VG). The VG acts as a pool of available storage for virtual disks. The physical extents from a volume group are concatenated to form one or more logical volumes (LVs). Physical extents are called logical extents (LEs) in the context of the logical volume they belong to. These steps ensure a flexible mapping of chunks from physical volumes to logical volumes.

Logical volumes have several advantages over conventional partitions. For example, logical volumes do not need to be contiguous, and logical extents can be mapped in arbitrary order onto physical extents of the backing storage. This greatly simplifies resizing and moving of logical volumes compared to partitions. Although partitions and logical volume managers have overlapping functionalities, they can still be used together. Partitions are often used as a basis for physical volumes, preventing operating systems from identifying the disk as empty, when lacking logical volume manager support.

2.3 Storage area networks

A storage area network (SAN) connects computers to remote storage, similar to how a local area network (LAN) connects computers to remote systems. SANs typically use fiber channel or Ethernet network infrastructures to provide block level access to remote storage: the same level as a physical disk. Specialized protocols run on top of the fiber channel or Ethernet, possibly using intermediate protocols like IP and TCP. The remote storage can consist of plain physical disks, but more often consists of virtual disks created from RAID arrays using an LVM. These levels of indirection give a lot of flexibility; storage can be consolidated and no longer needs to be directly connected to the machine.

A historically often used direct attached storage (DAS) interface in the enterprise market is the Small Computer System Interface (SCSI). Several ways exist to adapt this protocol to a SAN. The Fiber Channel Protocol (FCP) transports SCSI commands over fiber channel, while the Internet Small Computer System Interface (iSCSI) protocol does the same over TCP/IP. The terminology used by these protocols deviates somewhat due to the SCSI heritage. Storage servers are called targets, while clients are called initiators; consistent with respectively the interfaces and controllers on a SCSI bus. A single target can export multiple

disks, each identified by a Logical Unit Number (LUN). While originally a LUN would refer to a physical drive, in a SAN it more often refers to a virtual drive, constructed using RAID and or LVM. An important benefit of iSCSI over FCP is the ability to use existing Ethernet infrastructure.

The ATA over Ethernet (AoE) protocol encapsulates ATA commands in standard Ethernet frames. Advanced Technology Attachment (ATA) is a competing protocol to SCSI, primarily aimed at desktop systems. The benefit of directly using Ethernet instead of TCP/IP is a reduced protocol overhead. Calculation of TCP checksums for example puts a significant strain on the system; enough to try and offload these calculations to dedicated hardware. SANs based on iSCSI often employ TCP Offload Engines (TOE) or even full iSCSI controllers to improve performance, hardware not needed when using plain Ethernet. Similar techniques exist for SCSI based SANs; either encapsulate Fibre Channel over Ethernet (FCoE), or transport SCSI commands directly on Ethernet frames using HyperSCSI. The downside to using Ethernet is the inability to route the encapsulated protocol, which can be considered a security feature depending on requirements.

Contrary to AoE or iSCSI protocols, which encapsulate existing DAS protocols, the Linux Network Block Device (NBD) uses a custom protocol for communicating with storage. The NBD lives at the block device layer of the local operating system, forwarding I/O request over TCP/IP to remote storage servers. While the client kernel module is meant to run on a Linux system, the user-space server can run on any Unix based operating system. Alternatives to NBD are the Enhanced Network Block Device (ENBD) and Global Network Block Device (GNBD). ENBD introduces additional capabilities to the kernel module, for example multichannel failover; while GNBD is tuned specifically for shared disk usage together with the Global File System (GFS). NBD based storage integrates easily in a Linux environment, although it does not have the interoperability that AoE and iSCSI offer.

SANs can be used to share storage between multiple computers, possibly even with concurrent access. Traditional SCSI disks can be shared by attaching multiple controllers to the same SCSI bus. These shared disk semantics can also apply to SANs, with the added benefit of larger area networks allowing shared disks to be placed further away. Backups can be made to remote locations to prevent data loss in case of fire. Disks can be migrated to a different server platform when hardware fails, without the need to physically move the disk. Shared disk file systems can be used concurrently to share data between nodes. Even commodity hardware can be used to create shared disks.

2.4 Network attached storage

While storage in a SAN is shared at the block level, Network Attached Storage (NAS) is shared at the file system level. Disks are formatted with a file system of choice and then exported using a network file system. A SAN typically hides the remote nature of disk blocks from the operating system, but a NAS explicitly shares remote files. SAN protocols are implemented as drivers, while NAS

protocols are implemented higher in the operating system storage infrastructure. An inherent benefit of NAS is file-based access control: access can be granted on a per file or directory basis, similar to a regular file system. This makes NAS solutions better suited for simple file sharing than SANs, especially in consumer systems. SAN and NAS technology are however not mutually exclusive; an operating system can deliver NAS services based on SAN storage.

Current network file systems can be associated with the platform they were originally designed for. Server Message Block (SMB) and Common Internet File System (CIFS) originate from Microsoft Windows, CIFS being an evolution of the earlier SMB protocol. Network File System (NFS) can be used to export file systems found on Linux and other Unix based operating systems. The Apple Filing Protocol (AFP), originally part of the AppleTalk protocol suite, is a protocol for sharing files in Mac OS. Although different protocols, all belong to the family of network file systems.

Interoperability between network file systems and different operating systems is possible. A prime example is the Samba project: an open source implementation of the SMB/CIFS protocol. A similar effort exists for AFP. Together they enable Linux and other Unix based operating systems to share files with both Microsoft Windows and Mac OS. Mac OS in turn is capable of file sharing through SMB and NFS. To guarantee compatibility, most NAS appliances will support at least SMB/CIFS, NFS and AFS.

The abstraction of file systems in a Virtual File System (VFS) layer, makes transparent access to network file systems possible. A VFS allows uniform access to different concrete underlying file systems. An application can not distinguish between accessing a local file system or one accessed via NAS protocols. Even protocols not typically classified as network file systems, can be used to implement a VFS. In Linux for example, the File Transfer Protocol (FTP) and Secure Shell (SSH) protocol can be used to access remote files; they are typically implemented as a File system in User-space (FUSE). While the operating system is fully aware of the remote nature of some file systems, applications need not be.

2.5 Shared disk file systems

Shared disks require specialized file systems for concurrent access. A conventional file system will cause data corruption when used simultaneously on different computers. Uncoordinated instances of a file system can interfere with each other when updating disk blocks of shared file system data structures. This problem is often exacerbated by the use of caching. The system needs to prevent such concurrent modifications; often by using a distributed form of locking. Conventional locking of files, meant to prevent concurrent access by local applications, is insufficient. Those locks are implemented by the file system and hence subject to the same limitations.

The Global File System (GFS) is a shared disk file system developed by Red Hat. It depends on the Distributed Lock Manager (DLM) for managing concurrent access to shared disks. When shared storage is not yet available,

one can use the Global Network Block Device (GNBD) specifically designed for this purpose. The Oracle Cluster File System (OCFS) was originally created to store database files. It has now been extended to a more general purpose shared disk file system. Like GFS, it uses DLM as part of its cluster software stack. The Quick File System (QFS) from Sun Microsystems is often used in conjunction with their Storage Archive Manager (SAM). SAM can move data between different types of storage based on usage: a form of Hierarchical Storage Management (HSM). QFS uses a single meta-data server to coordinate concurrent access.

Shared disk file systems use cluster type software to solve concurrency problems. GFS and OCFS use the DLM distributed lock manager, which comes as part of a cluster stack. QFS uses the Sun Cluster system. A common problem for clusters and subsequently shared disk file systems is detection of failed nodes. It is important that nodes that appear to have crashed can be prevented from resuming later on. This is called fencing and requires hardware support. When no cluster stack is available, the above file systems can still be used as standalone file systems.

2.6 Distributed file systems

Distributed file systems differ from shared disk file systems in the way data is shared. In a shared disk file system, data is shared at the block level by using shared disks; the shared disk file system has a coordinating function. In a distributed file system, data is shared at a higher level by the distributed file system itself; block storage or file systems are only used to store data. In short, a distributed file system shares data stored on disks, while a shared disk file system stores data on shared disks.

Lustre is a distributed file system maintained by Sun Microsystems. It splits the file system into meta data and file data. Meta data is stored in a single meta data target (MDT), while file data is stored in one or more object storage targets (OSTs). Both types of targets are backed by standalone file systems. A server providing access to the MDT or OSTs is called a meta data server (MDS) or object storage server (OSS), respectively. High availability is achieved by using shared disks, allowing an MDT or OST to be shared between two servers. The servers are then configured as an active/passive pair: the passive server takes over when the active server fails. In practice, the MDT is always shared: losing access to the MDT, means losing access to the complete distributed file system. An OST is not always shared: losing access to an OST, only means losing access to the files stored on that OST.

GlusterFS is a distributed file system based on translators, an idea inspired by the GNU/Herd operating system. Clients access storage on the servers through a series of translators, which determine the behavior of the system. Translators are specified in a volume file and can be made to run on both clients and servers. GlusterFS uses POSIX based file systems as a starting point to store files: directories can be imported using the posix translator. There are lots of additional translators available: to serve or access volumes over a net-

work, distribute files over multiple servers, read ahead or write behind, stripe or mirror volumes, fail over between servers, etc. It is left up to the user to write interoperable volume files for both clients and servers.

Since distributed file systems share data at a higher level, often above the file system level, it is possible to use other storage techniques as backing storage. As mentioned earlier, Lustre uses shared disks to increase availability. GlusterFS could potentially use shared disk file systems to improve availability of files when a server goes down. Most distributed file systems will also use RAID or LVM at some point to increase reliability and flexibility. It really shows how storage systems can be stacked.

3 Use cases of a virtual shared disk

This section presents use cases for a virtual shared disk. Since it can replace conventional shared disks, we look at some typical applications. A virtual shared disk should be able to function correctly in these settings. First is simple network sharing of block storage in section 3.1. Then comes performance improvemens by using multipath in section 3.2. Finally section 3.3 shows a practical application of shared disks to improve availability of services.

3.1 Shared disk

A virtual shared disk can replace hardware solutions for shared disk applications: storing standalone file systems, shared disk file systems, distributed file systems, etc. Sharing access to storage improves data availability on server failure. For standalone file systems this means another server can mount the file system, while for shared and distributed file systems it means other servers can maintain access to the file system. This is no different for virtual shared disks than it is for hardware shared disk solutions; virtual shared disk software can act as a drop-in replacement for shared disk hardware.

The added benefit of virtual shared disks is storage reliability. Single shared disks can reduce the impact of server failure, but do not protect against storage failure. Redundant hardware solutions exist but are generally expensive. A virtual shared disk can be constructed using commodity hardware, removing the need to rely on proprietary implementations. This is also true for underlying logical volume management and RAID implementations: proprietary implementations can complicate data recovery. Virtual shared disks protect against both server and storage failures using commodity hardware and software.

3.2 Multipath

Virtual shared disks have inherent parallelism which can be exploited for performance and reliability. Hardware-based shared disk solutions can have both internal and external redundancy. Internally there is redundant storage to prevent data loss, while externally there is redundant access to prevent communication failure. Both reliability and performance are increased by instructing higher level protocols to use the multiple external access ports. This technique is called multipath: storage commands are distributed over ports to increase performance, or can fail-over to other ports to increase reliability. Virtual shared disks can be accessed in the same parallel manner.

A virtual shared disk distributes data over multiple locations. This in a sense combines the internal and external redundancy of hardware solutions: each location acts as both redundant storage and access port. SAN and NAS techniques can exploit this parallelism to increase reliability and performance. For example, iSCSI can use multipath techniques to access virtual shared disks at the SAN/block level. Parallel CIFS and NFS solutions in combination with a shared disk file system or distributed file system can exploit the parallelism

at the NAS/file level. The inherent parallelism of virtual shared disks combines naturally with higher-level protocols.

3.3 High availability cluster

Concurrent access to shared disks needs to be coordinated. Although shared disks allow multiple servers to access data in parallel, doing so might corrupt the overall data structure. Stand alone file systems for example are not meant to be used concurrently, while shared disk and distributed file systems use locks to prevent concurrent access where it is unsafe. When a server fails, all its locks need to be released to allow fail-over to another server. There also needs to be a guarantee that the failed process does not resume operation later on. The problem becomes one of detecting failed processes.

A high availability cluster provides the necessary infrastructure for fail-over. In an asynchronous network, like Ethernet, it is impossible to distinguish between failed and slow processes. To get a form of failure detection, a high availability cluster uses heartbeat messages: if a server stops responding to heartbeat messages it is considered to have failed. There is however no guarantee of failure: the server could just as well have been slow. To prevent the server from resuming later on, it is fenced. Fencing means the cluster removes the presumed faulty server from the system by for example cutting its power, or instructing the storage system to deny further access. Hardware support is required for reliable fencing.

Fencing can lead to problems when the cluster splits up. It is possible for parts of the system to get isolated from one another due to network partitions. This situation is called a split-brain: the different parts continue to operate and start to make possibly conflicting decisions. Partitions can for example start fencing each other, bringing the whole cluster down. A virtual shared disk implemented using the cluster infrastructure, could receive different writes for each partition. It is very difficult to decide which version of the disk to use when the cluster partitions are recombined. To prevent these problems from happening, a quorum is used: only partitions containing a majority of cluster nodes can make decisions.

A virtual shared disk can operate independent of the cluster infrastructure. Applying RAID to shared disks would require special fail-over facilities to prevent concurrent access to RAID members. Virtual shared disks natively incorporate quorum and can do without failure detection. This is because the consensus needed to determine failed nodes is a stronger primitive than the sequential consistency needed to allow concurrent access. The benefit is that virtual shared disks can be used concurrently in a high availability cluster.

4 Survey of related work

This section explores the theory needed to construct a virtual shared disk. It also looks at the solutions used by competing implementations. Central to the construction of a virtual shared disk is the impossibility of deterministic consensus when processes can fail. Section 4.1 looks at the problem in more detail. Section 4.2 investigates why this issue does not seem to affect atomic registers in their ability to provide shared access to a single value. A promising avenue for construction of a virtual shared disk.

But what kind of behaviour should we expect from a virtual shared disk? Section 4.3 looks at models for different types of consistent behaviour, followed by section 4.4 describing the choice of model made for virtual shared disks. This leaves section 4.5 to investigate competing solutions for shared disk semantics.

4.1 Distributed consensus

Deterministic consensus is impossible in an asynchronous distributed system with even one faulty process. This important theorem is proved by Fischer and others [20]. There are however other methods of reaching consensus in a distributed system, generally by changing the assumptions of the impossibility proof.

For example partial synchrony [18] assumes some unknown or future upper bounds exist for message latency and relative speeds of processes. In some cases failure detectors [1] can be used to distinguish between slow and faulty processes. One can also weaken the property of guaranteed termination to probabilistic termination [12].

Contrary to atomic registers however, deterministic consensus cannot be made fault-tolerant in an asynchronous system.

4.1.1 Impossibility of deterministic consensus

A fundamental problem in distributed computing is that of reaching agreement between asynchronous processes. Take a system of processes, with divergent input values, and make them decide on a common output value. A consensus protocol solving this problem must be consistent: all deciding processes agree on the same value. The protocol must be terminating: all correct processes must decide. And finally the protocol must be non-trivial: the decision should depend on the input. The proof of Fischer, Lynch and Paterson [20] shows the impossibility of such a protocol when confronted with even one faulty process.

Consider a system of processes trying to reach consensus over a boolean value. Processes can decide 0 or 1. The combined states of the individual processes and message channel define the configuration of the system. A transition from one configuration to the next is achieved by applying events: the reception of a message by a process and the resulting response. The receiving process sends a finite number of messages and possibly decides on a value, in either way ending the transition. Configurations are 0-valent when, starting from the

configuration, only 0 can eventually be decided; 1-valent when only 1 can be decided. When both 0 and 1 can still be decided, the configuration is called bivalent.

Assume a protocol exists for reaching deterministic consensus in an asynchronous system. A proof by induction will show a contradiction when even one process is allowed to crash. As the base case, it can be shown that there is a bivalent configuration in the group of initial configurations. For the induction step pick an event, then we can divide reachable configurations in two groups. One group will be reachable by applying the event at some point, the other without applying the event. The first of these groups will again contain a bivalent configuration.

Starting from a bivalent initial configuration, it is possible to only apply events that transition to another bivalent configuration. Pick an event that will be delayed indefinitely because of a process crash. According to the induction-step, there is a sequence of events after which applying event e would yield a bivalent configuration. Apply the first event of this sequence to reach another bivalent configuration.

The system can confine itself to bivalent configurations: no decision is ever made. This contradicts the termination property of the assumed consensus protocol. There is no deterministic one-crash resilient consensus protocol.

4.1.2 Paxos

Paxos [32, 30] is a protocol for reaching probabilistic distributed consensus using clocks. To reach agreement, votes are cast in multiple rounds, called ballots. Each ballot requires a unique number, a quorum that will overlap with other ballots, and a decree identical to the most recent decree a quorum node previously voted for. When no previous decree is available a new one can be created. A decree is accepted when all nodes in a quorum vote for it.

Ballots consist of roughly two phases. During the ballots, nodes keep track of the last ballot number started, the last ballot number received, and the last vote cast. A vote consists of a node's identity, a ballot number and a decree. Each ballot number can be extended with the identity of the node that created it. This prevents duplicate ballot numbers when multiple nodes concurrently start a new round. Majorities are used to ensure quorums of ballots have at least one node in common.

A round starts with a node choosing a new ballot number, greater than the one it used last time. Then it sends a message containing this new number to all or possibly a subset of nodes. Nodes only respond to the message when the ballot number is larger than the last ballot number they received. If this is true, the node will also record the ballot number. The response contains the ballot number and the last vote the node cast in older ballots. Eventually the initiating node will learn of the previous votes cast for a majority of nodes in the system.

When a majority has responded, the initiating node can start the actual ballot. It sends its decree and ballot number to the other nodes, using the same

majority as its quorum. The decree must be identical to the last vote cast by any of the quorum nodes. If no votes have been previously cast, the node can choose a decree of its own. Nodes only respond to the message when the ballot number equals the ballot number recorded earlier. If this is true, the node will also record its vote. The response contains the ballot number and the identity of the node. Eventually the initiating node can receive votes from a majority of nodes in the system.

When the quorum has voted for the decree, the initiating node records the decree and informs all other nodes of its success. There is however no guarantee that all the nodes in the quorum will respond: nodes might have crashed. The protocol, as described, ensures safety but not liveness. Ballots have to be started repeatedly until one succeeds. Too many concurrent ballots can however prevent progress.

Initiating a ballot with a high ballot number, will prevent nodes from participating in previous ballots. This can prevent any previous ballot from succeeding. To prevent this from happening, timers are used to start new ballots when the previous one did not reach its quorum. Based on maximum message and processing delay, they limit the rate at which ballots can be initiated. Additionally a president can be chosen to initiate ballots, preventing concurrency.

4.2 Atomic registers

Traditionally access to shared data structures is controlled using locks. Locks prevent concurrent operations that could otherwise interfere with each other: like reading while writing. This however, gives rise to a problem of mutual exclusion, where one process blocks access for all other processes. Peterson [38, 39] shows how concurrent access to large data structures can instead be made wait-free. Wait-free means that operations take a finite numbers of steps, independent upon the speed of other processes. This prevents a process from locking out others.

The simplest of shared data structures is the shared register, which stores only a single variable. Lamport [31] describes how one can construct such a register and what properties it might have. These properties can be strengthened by implementing new registers on top of weaker ones. The proposed atomic registers are however single-writer and no atomic multi-reader register is constructed.

Atomic registers behave as if there is a sequential order on operations: as if all concurrent access can be linearized. The problem of an atomic single-writer multi-reader register is solved in [4] for exponential time and improved upon in [43] to polynomial time. Other solutions are presented in [38, 7, 36].

A solution for atomic multi-writer multi-reader registers is first proposed in [45] but found to be incorrect. Later solutions include [39, 34, 15]. An optimal atomic multi-writer multi-reader register is constructed by Israeli and Shaham [28] using precedence graphs.

Although atomic registers are wait-free, they are still susceptible to unavailability of underlying registers. Wait-freeness only applies to the relative speed

of readers and writers using the constructed register, not to the speed of the registers it is constructed on. This is especially important when reader and writer processes also implement the underlying registers; a slow register could stall other processes, negating the benefits of a wait-free algorithm. Attiya and others [6] show how an atomic single-writer multi-reader register can be constructed, while only communicating with a majority of processes. This enables wait-free algorithms, based on single-writer multi-reader registers, to run in distributed systems with faulty processes. For a similar technique using the more general concept of quorum see Herlihy [24], who uses overlapping process groups to replicate objects. When a regular quorum is not available, ghost processes [44] can be temporarily introduced to maintain availability. Since atomic registers can be made fault-tolerant using majorities or quorum, so can the registers constructed on top of them.

The performance of atomic single-writer registers can be improved by limiting the number of readers. Dutta and others [17] show how both reads and writes can be accomplished in one communication round-trip for single-writer registers, and why this is impossible for multi-writer ones. Atomic registers where reads and writes only take one round-trip are called fast. So called semi fast [22] implementations do not limit the number of readers: reads try to use one round-trip, but can fall back on a second round-trip when unsuccessful. The resulting trade-off between fault-tolerance and performance is further studied by Georgiou and others [21].

4.2.1 Single-writer multi-reader register

Wait-free algorithms based on atomic registers in shared memory can be emulated in message passing systems. Attiya, Bar-Noy and Dolev [6] show how to construct a crash-resilient wait-free single-writer multi-reader register for this purpose.

Processes implementing this register, communicate by sending a message to all participating processes, including the sending process itself. They then wait for acknowledgments from a majority of processes. The communication procedure uses a ping-pong mechanism to ensure only one message at a time is in transit between two processes. This ensures received acknowledgments refer to the last message sent. An acknowledgment may contain additional information.

To write a new value the process communicates this value to a majority of processes. Each write is labeled, enabling a process to recognize and store only the most recent write. For simplicity, a write is identified with its label instead of its value.

To read from the register, a process requests the labels from a majority of processes. It returns the most recent write among the results, but not before communicating this value to another majority of processes. Else it would be possible for a read to return the label of a concurrent write, and a subsequent read to return an earlier label.

Both read and write operations terminate if and only if their communications

complete. This is the case when a majority of processes acknowledge the communication. Since non-faulty processes always acknowledge incoming messages, this majority is sufficient to make the algorithm wait-free.

4.2.2 Multi-writer multi-reader register

Using single-writer multi-reader atomic registers, it is possible to construct a multi-writer multi-reader atomic register. One solution by Israeli and Shaham [28] builds a precedence graph using information stored in single-writer multi-reader registers. Such a graph serializes write operations and can subsequently serialize read operations, modeling the behavior a multi-writer multi-reader register.

A precedence graph is a directed in-tree: edges point towards the root of the tree. The root is a special virtual node; all other nodes are stored in registers. A directed edge in the graph from node b to node a, indicates that a precedes b. However, when an edge from node c to node a is added, precedence between b and c is undefined.

The algorithm stores nodes and edges of the precedence graph using single-writer multi-reader atomic registers. Besides the inherent identity of the writer, each node has been assigned an address. A register can then be used to store a label: a combination of a node and an outgoing edge. The node is defined by its address. The outgoing edge is defined by the identity and address of the node it points to. Notice the identity of the node's writer is stored implicitly; a consequence of using single-writer registers.

The notion of local precedence is used to determine precedence when two nodes have the same parent node. When the nodes are created by the same writer, the older node takes local precedence. For nodes from different writers, the node with the highest identity has local precedence. A frontal branch can then be defined by recursively following edges from the root towards the leaves: always choose the node that is locally preceded by all other child nodes. The leaf node of the frontal branch is called the last node of the graph.

A precedence graph can be used to construct a simple sequential register. Only consecutive read and write operations are considered for such a register. Registers are initialized with labels pointing back to themselves, which results in a frontal branch that is limited to the virtual root node.

The sequential write protocol starts by collecting the labels from all the registers, and constructing the precedence graph. A new node is added to the graph by connecting it to the frontal branch: its edge points to the node with the largest identity in the branch, smaller than the identity of the writer. Notice that this creates a new frontal branch.

The sequential read protocol collects the labels from all registers, and constructs the precedence graph. Then it simply returns the last node of the frontal branch.

Using the sequential protocol in a concurrent setting will fail. An unused address at the start of a write, might have become referenced at the end of the write. This can result in an incorrect precedence graph: an old node is

preceded by the new node. To prevent this violation of the sequential order, the concurrent write protocol stores two labels per register. A current label equal to that of the sequential algorithm, and a new label to stage a pending write.

The concurrent write protocol starts by constructing the precedence graph from the current labels in the registers, similar to the sequential algorithm. When choosing a free address for the new node, edges of both new and current labels are considered. This ensures that no current or pending label has an edge referencing the new node. The new node is first written as a new label. The write protocol subsequently reads the register of the referenced node, and checks if it has changed. The node referenced by the new label might have been overwritten. The current label is then updated: if no changes were detected, it is updated to the new label, else to the new label modified to reference itself.

The concurrent read protocol is less intuitive. The protocol starts by consecutively constructing three precedence graphs. The first and third graphs are constructed normally: by sequentially reading the registers, starting from the lowest, going up to the highest. The second graph however, is constructed backwards: starting from the highest register, going down to the lowest. Due to concurrency, it is possible none of the three graphs are consistent. The three graphs are therefore compared to determine a correct response.

4.3 Consistency models

In an asynchronous system there is no global clock to serialize operations on shared objects. Operations still need to behave in some predictable manner for the system to be of use, especially in the face of concurrency. Such behavior can be described using a consistency model: a guarantee that the system behaves according to certain rules. Different consistency models are available to define consistent behavior for a distributed system.

Operations in distributed systems are not instant but are executed during an interval: the time between invocation and response of an operation. Overlapping intervals indicate concurrent operations, while non-overlapping intervals indicate sequential operations. The absolute timing of these operations is however not available to individual nodes in an asynchronous system; nodes can adopt competing views due to message delays. Some coordination is required to ensure consistent behavior.

The behavior of a system can be justified as consistent using a history: a global order for invocation and response events. Operations should behave in accordance with a history and maintain object semantics. For example, items from a stack object can only be popped when they have been pushed earlier. To simplify reasoning about object semantics, histories are often limited to sequential operations.

In a sequential history, operations can be considered atomic. Lack of concurrency ensures pending operations are invisible: operations seem to take effect instantaneously. In a sequential history every invocation event is directly followed by the corresponding response event. This requires some reordering of events, relative to their absolute time occurrences.

Often multiple histories are available to justify system behavior. For example, the value of a shared variable is only important on a read operation; preceding write operations can be ordered arbitrarily when they are overwritten before the next read. A consistency model only states which histories are acceptable. It is not necessary to pick a specific one.

Consistency models can limit the amount of reordering allowed to create a sequential history; reordering is relative to the original order of events in absolute time. An example is to not reorder events of originally sequential operations. The exact limitations depend on the specific consistency model. How a system manages to comply with the consistency model is of no concern to the model.

There are different consistency models to pick from when designing a system. Some examples are: strict consistency, sequential consistency, linearizability, quiescent consistency, serializability and causal consistency. We will look at these models in more detail:

Strict consistency requires operations to take effect immediately, consistent with their occurrence in absolute time. A natural fit for synchronous systems, where operations can be executed in the order they were invoked; but too much of a constraint for asynchronous systems, where lack of a global clock prohibits implementation. Strict consistency can reorder response events to remove concurrency.

Sequential consistency [29] requires operations to be executed in their original program order for each separate process. This removes the need for a global clock, since operations from different processes are never compared using absolute time. As a result, sequential operations from different processes can behave counter-intuitive when processes partially synchronize. Sequential consistency can reorder events when they belong to different processes.

Linearizability [25] requires the partial order of originally sequential operations to be preserved. This is called atomicity for read and write objects. Linearizability is a stronger property than sequential consistency, which only preserves this partial order per process. Contrary to sequential consistency, linearizability is a local [46] property: if operations for each separate object are linearizable, then all operations are linearizable. Linearizability can reorder events when they belong to concurrent operations.

Quiescent consistency [5, 42] requires operations separated by quiescence to remain in their original order. Quiescence is a lack of pending operations on an object: all invocation events have had a corresponding response event. This consistency model differs from linearizability and sequential consistency in that program order is not necessarily preserved. Quiescent consistency can reorder events belonging to operations not separated by quiescence.

Serializability [37] requires sequential consistency of transactions. When transactions are also linearizable, this is called strict serializability. Transactions are atomic combinations of operations, possibly on different objects. Sequential consistency and linearizability can be seen as special cases of serializability and strict serializability, when transactions are single operations on single objects. Serializability can reorder events when they belong to different processes. Strict serializability can reorder events belonging to concurrent transactions.

Causal consistency [3] requires only causally related operations to remain in their original sequential order. There is no global history: operations that are not causally related do not have to be consistently ordered. Different processes can see these operations in different orders. There is no total ordering for events. Causal consistency can reorder causally unrelated operations differently for each process.

4.4 Shared disk semantics

Simulating a shared disk in a distributed system requires shared disk semantics. In other words, the system should behave like a singular shared disk, while actually being distributed. Linearizability is the preferred consistency model for this application. It ensures correct behavior when multipath techniques are used: sequential operations arriving on different paths act consistently. In addition, the locality of linearizability eases implementation of larger block devices: each block can be handled individually without breaking consistency. Linearizability is scalable and supports multipath.

Multipath is important for distributed shared disks, since it exploits the redundancy of the system. Sequential consistency could break shared disk semantics: a write on one path, can be invisible to a subsequent read on another path. Similar problems exist for causal consistency: causal relations external to the system cannot be detected and can cause operations to appear in different orders depending on the chosen path. Quiescent consistency can break shared disks semantics even without multipath being used: concurrent operations can prevent sequential operations from taking effect. Multipath needs to be explicitly supported to maintain shared disk semantics.

Linearizability of a virtual shared disk can also be modeled using strict serializability. Since an operation on the shared disk is replicated to multiple locations, these replicated operations can together be seen as a transaction. Success of such a transaction does not necessarily require the success of all component operations. Exact requirements depend on the implementation: it is possible only a majority of operations needs to succeed. In short, external operations on the shared disk are linearizable, while internal operations are strictly serializable.

4.5 Distributed storage

There are competing solutions for distributed storage available, solving similar problems to those of virtual shared disks. The following sections describe these solutions and compare them to the virtual shared disk implementation.

4.5.1 DRBD

The Distributed Replicated Block Device (DRBD) [40, 19] is used as a replacement for shared disks in high availability clusters. By mirroring writes between local and remote disks, it ensures reliability in the event of node failure. A

typical application is the storage of shared disk file systems, where it assumes a distributed lock manager will prevent concurrency. While originally developed for only mirroring two nodes, more nodes can be supported by stacking the implementation on top of itself.

Special precautions are taken to prevent mirrors from diverging in the event of concurrent writes. The developers analyzed all possible orderings of messages in the write protocol. Asymmetric orderings can be detected by the protocol, and used to determine which write goes first. Symmetric orderings rely on one of the mirrors being instructed, up front, to discard symmetric concurrent writes. Shared disk semantics are not guaranteed under concurrency.

Nodes can run either as primary or secondary. Primary nodes accept read and write requests, while secondary nodes only accept replication requests. This is a natural fit in high availability frameworks, which on node failure will restart a service on a backup node. The backing storage can then be switched from secondary to primary on the backup node. An alternative is to run both nodes as primary, removing the need to explicitly switch roles. This however prevents the framework from protecting against concurrency. Isolated nodes, simultaneously running as primary can cause mirrors to diverge.

The situation where two primary nodes are isolated from each other is called split-brain. It means each node can process writes independently, causing the mirrors to diverge. A bitmap is used to store unsynchronized writes. When the mirrors reconnect, DRBD needs to decide on a synchronization strategy. It is possible to merge the mirrors when writes do not overlap, or to choose a mirror based on its age measured in unsynchronized writes. As a last resort the decision can be left to an administrator.

DRBD does not support concurrent writes to two primary nodes, therefore multipath can only be used for failover purposes. Concurrently writing over two paths can have unexpected results and even failover can become problematic if paths are switched too fast. Virtual shared disks do support concurrency and guarantee storage integrity. The requirement of a majority of reachable processes for virtual shared disks also ensures no split-brain situation can ever occur.

4.5.2 Chubby

Chubby [8] is a lock service for loosely coupled distributed systems. It acts as a distributed file system and can for example be used to store names, advertise small results or distribute configuration files. More importantly, it can be used to acquire locks. Chubby employs Paxos to ensure fault tolerant consensus in asynchronous networks. The file system interface combined with locks ensure familiarity to programmers.

In practice, locks are coarse-grained and primarily used to elect a master node. This application specific master node can in turn handle any fined-grained locking if needed. Leases are used to eventually revoke locks from slow or crashed nodes. They take into account a worst case message delay: master lease times are larger than their client counterparts to ensure mutual expiration. Keepalive

messages are used to renew the lease times of locks.

Chubby is a general locking service, which can be shared for use by other services. Googlefs [23] and Bigtable [10] for example use Chubby to elect a master. Support for events and caching is available to reduce the load on a Chubby cell. It is also possible to distribute the name space over multiple cells, or to proxy and aggregate requests. This allows for improved scaling of the system.

Coarse-grained locking makes chubby unsuitable for reaching consensus on each individual read or write operation. It could be used instead to elect a single master, which decides on a consistent ordering, but there is no guarantee failing master will be replaced quickly. The whole point of chubby is to use it as a basis for systems with less restrictive consistency models as demonstrated by Googlefs and Bigtable.

4.5.3 Petal

Petal [33] is a distributed storage system. It abstracts from distributed physical storage, to single virtual disks. Three directories are used for this abstraction: the virtual disk directory, the global map, and the physical map. The virtual disk directory translates virtual disks to global maps. Global maps translate virtual offsets to storage servers. Physical maps translate virtual offsets to physical disks and offsets, for specific global maps. These three abstraction levels enable a flexible translation from virtual to physical disk addresses.

The virtual disk directory and global map are updated using Paxos, ensuring a consistent state for all involved nodes. Reconfiguration of the system can therefore proceed as long as a majority of processes is available. The storage itself is not necessarily fault tolerant; it depends on the chosen redundancy scheme. Petal currently supports striping and chained declustering [26]. Striping does not provide any redundancy, while chained declustering allows for single failures. Redundant reading and writing of data, does not require the use of Paxos.

An added benefit of chained declustering is load balancing. Chain declustering mirrors data blocks by storing them alternatively on one of two adjacent nodes. In case of failure, the load is distributed over these two neighboring nodes, which in turn can do the same with their neighbors. This is contrary to a standard mirror, where only one node is available to take over the load, effectively halving system performance. The downside to chained declustering is a higher risk of data loss. With chained declustering, data is at risk when one of two adjacent nodes fail. In a standard mirror data is at risk only when one specific mirror node fails.

To ensure consistency of data, one mirror of the data is designated primary and the other secondary. Only the node with the primary copy can write data, and will lock both copies while doing so. This strict order prevents deadlocks: a secondary node will make sure the primary is down before writing data. Stale and busy flags, stored on stable storage, are used to recover from node failure later on. Read requests can be fulfilled by either node. Clients will alternatively

try the primary and secondary node until one succeeds or both fail.

Pedal does not support network partitions; it has no way of recovering from split-brain situations. Split-brain will never occur for virtual shared disks, since a majority is needed for operations to succeed. Also, failover in Pedal requires the secondary node to ensure the primary is down. This is a problem in itself, since one can not distinguish between slow and failed nodes in an asynchronous network. The inability of detecting failed nodes is what makes implementing a virtual shared disk difficult.

4.5.4 Federated Array of Bricks

A Federated Array of Bricks (FAB) [41] is a distributed disk array. The system is designed to deliver reliable storage on primarily commodity based hardware. It uses quorums to implement strictly linearizable data access. A Paxos based protocol is used to resolve concurrent reconfiguration. Strict linearizability ensures single disk semantics are preserved, even in the face of failing writes. This opens the possibility for operations to abort.

A few key data structures are used to translate logical to physical disk addresses. A volume layout maps a logical disk and offset to a segment. The segment group in turn maps the segment to storage nodes, conform the chosen storage layout. A disk map finally stores the physical disk and offset for actual pages of the logical disk. Timestamps, unique to each node, allow for consistent access to the logical disks. When a node encounters a concurrent update with a higher timestamp, it will abort.

The volume layout and segment groups are replicated using dynamic voting [15]. This Paxos like algorithm is used for consensus on reconfiguration transitions. The disk map and timestamps are managed locally. While the volume layout, segment groups and disk maps are stored on disk, NVRAM is used for storing timestamps and buffer cache.

FAB supports both replication and erasure encoding, to reliably store data segments on multiple nodes. A write request to these nodes, is handled in two phases. In the first phase, the data is queued at the nodes; in the second phase, the queued data is committed to disk. Timestamps are stored for both the queued data and the on disk data, allowing the algorithm to detect incomplete writes. A read request finishes in a single phase when no incomplete writes present themselves. Otherwise, the request will require an additional two phases for finishing the incomplete write with a higher timestamp.

The timestamp table requires garbage collection to reduce it size. If a timestamp would be stored for each disk block, the table would require too much NVRAM. Timestamps are used to distinguish concurrent updates and to recover from failures; they can be removed when all replicas of a data segment respond after an update. Since old timestamps can still arrive with delayed messages, a node will wait for a short period before actually removing the entry. This period is chosen conservatively based maximum clock skew and scheduling delay. Another option is to store timestamps for ranges instead of individual

blocks. Writes generally update multiple blocks, which enables a tree like organization of the timestamp table.

Separate NVRAM as used by FAB is not commonly found in commodity hardware. Virtual shared disks have no such special requirements. FAB also requires synchronized clocks, for which the network time protocol (NTP) can be used. If clocks are not synchronized, a lot of unnecessary aborts will occur. In comparison, timestamps for virtual shared disks are built into the algorithm itself. There is no requirement for external synchronization.

4.5.5 TickerTAIP

TickerTAIP [9] is a parallel raid architecture that distributes a traditional RAID array over multiple worker nodes. Communication of data blocks is minimized by reading the minimal amount of blocks necessary to calculate parity blocks. No data blocks need to be read for full stripe writes, where all data is replaced; but some data blocks will have to be read when stripes are partially modified. The latter can be achieved by either reading the data blocks that will be modified, or the data blocks that will not be modified. Both methods can be used to calculate parity. Choosing the method requiring the least amount of reads will give the best performance.

Another consideration is where to calculate parity blocks. This can be done either at the originator of the request or the node containing the parity block. Partial calculations can be done at the nodes with data blocks, and then results can be shipped off to the node storing the parity block. Using nodes to calculate partial results, will ensure the load is balanced over as many nodes as possible.

Write atomicity is ensured using a two phase commit. Writes should either succeed or fail. Data is replicated to enough nodes to ensure the system can at least survive a single failure; it should be possible for other nodes to restart and complete partial writes. Only when replication succeeds, is the write committed, else it aborts without making any changes.

Some form of serializability is needed to guarantee consistent behavior of the system, especially in view of concurrency and failures. For this purpose TickerTAIP uses request sequencing. This is based on an directed acyclic dependency graph. The graph expresses the dependencies between requests, making sure aborted requests propagate to requests depending on them, which in turn will also abort. Both explicit dependencies, defined by the user, and implicit dependencies, defined by the system are supported. Implicit dependencies use an arbitrary serializable schedule.

The sequencer uses a state table to record the status of requests based on their dependencies. This table can be either managed centrally or distributed. In the central case a single sequencer stores the state table, while a backup is used to prevent a single point of failure. The distributed case requires a distributed consensus protocol to ensure a consistent state table on all nodes.

TickerTAIP shifts the burden of consistency onto a sequencer and requires a distributed consensus protocol for distributing this sequencer over multiple nodes. There is however no guarantee such a protocol will reach consensus in

an asynchronous network. Virtual shared disks do not depend on consensus to function.

4.5.6 RAMBO

RAMBO [35] is a reconfigurable atomic memory service for dynamic networks. For reconfiguration it uses the Paxos protocol: to reach consensus on the sequence of configurations. Read and write operations are handled using overlapping read and write quorums. Each node stores a list of configurations active in the system. A background gossip protocol ensures nodes eventually get informed about new configurations.

Read and write operations are executed for all known configurations. Only when a fixed point is reached will the operations finish. Fixed point means that a quorum is reached for all participating configurations. First a query is sent to a read quorum of all active configurations. Then the new value is propagated to a write quorum of all active configurations. Nodes can learn of new configurations during the query phase, and decide to restart.

Garbage collection is used remove old configurations. A process can start a garbage collection when it is aware of two consecutive configurations in the system. First a read and write quorum of the old configurations is informed of the new configuration. Then a write quorum of the new configuration is informed of the latest tag. Configurations are collected sequentially, one at a time, only when all previous configurations have already been collected.

Consensus on new configurations is reached by using a global consensus service: one for each new configuration. This service is implemented using Paxos. New configurations can be suggested by members of the previous configuration. A separate reconfiguration service initiates requests to the consensus service and subsequently informs the system of the results.

The focus of RAMBO is on reconfiguring quorum configurations. It does not solve the problems associated with strict linearizability: handling partial writes of failed processes. Overlapping read and write quorums are used by RAMBO for linearizability but not strict linearizability. Virtual shared disks could however be complemented by the reconfiguration system of RAMBO.

5 Asynchronous virtual shared disk algorithm

We start by looking at a preliminary algorithm for virtual shared disks. This algorithm can cope with asynchronous networks but not failing processes. It is used as an introduction to modeling algorithms in the Promela language for use by the Spin model checker; and implementing the algorithm in C for use on the MINIX operating system.

The algorithm uses two types of nodes: sources and targets. Applications running on sources create read and write operations for the system to process. Physical storage on targets is used to store data for the virtual shared disk. The protocol runs on both the source and target nodes and is outlined as follows:

1. Sources send operations to all targets.
2. Targets place the operations in a queue.
3. Targets acknowledge by sending back the complete queue.
4. Sources place the queues from all targets in a view.
5. Sources use the view to decide, and inform targets of a commit order.
6. Targets commit the operations to disk.

It is very important that all sources reach the same decision; else, operations may be committed in different order at different targets. Sources do not always have enough information to reach a decision though. That is why targets will continue to send queue updates to a source, as long as operations from said source are in its queue. This way sources will eventually gather enough information to decide on a consistent commit order.

Imagine a situation where sources 1 and 2 simultaneously send an operation to targets A and B. Operations from source 1 are labeled with 1, operations from source 2 with 2. The views created by A and B, directly after the operation is queued, might look as follows for source 1 and 2 respectively:

A	B
1	2
	1

A	B
1	2
2	

The columns of these views, represent the queues at the targets. To be more precise: a snapshot of the queue when the operation was acknowledged. Apparently the operations have overtaken each other in this example. Target A was reached first by operation 1, while target B was reached first by operation 2.

This situation is however not apparent to the sources: they can not distinguish between operations being queued or being committed. Source 1 for example can not determine whether operation 2 has already committed at target B, or still needs to be queued. The sources will therefore wait for more

information. Eventually the targets will send another update of their queue, after which both sources will see:

A	B
1	2
2	1

The group of operations 1 and 2 is called a conflict group, since one of these operations has overtaken the other. A conflict group can be easily recognized by imagining lines in the view between identical numbers. Crossing lines indicate conflicting operations which can be added to the conflict group. In this example operations 1 and 2 are conflicting.

After determining which writes belong to a conflict group, we can determine a commit order. For example by choosing that the lowest numbered operation goes first. In this case operation 1 is committed first, followed by operation 2. Notice that a conflict group should always have the same operations in each column. This way all sources will reach the same decision.

An alternative result for the concurrent write, could have been the following view for source 1:

A	B
2	2
1	1

In this case there are two separate conflict groups, each containing only a single operation. Source 1 will decide to first commit operation 2, and then operation 1. Source 2 could have given an immediate commit after queuing its operation.

There is a problem, skipped over by the previous examples. Using a single sequence number per source is enough when considering only a single operation per source. But when a source is capable of subsequent operations, more sequence numbers are needed. Even when a single source will never have multiple outstanding operations.

Take the following view for source 1:

A	B
1	2
2	1

There is no guarantee that both operations labeled with 2 in this view, are copies of the same operation. The 2 in column B could be part of an older operation already committed on target A. Then the 2 in column A would be part of an operation not yet queued at target B. This confusion could lead to sources constructing incorrect conflict groups. Sources could reach conflicting

decisions and targets could commit the wrong operations. The solution is to give each source two sequence numbers.

The sequence numbers are not needed for a source to track its own operations. Sources only have one outstanding operation at a time. When a source starts a new operation, none of its older operations will show up in the view. This is due to the FIFO property of the communication channels between sources and targets. An operation from a source even acts as a kind of barrier, starting the moment it is queued at a target. As long as its operation is in the queue, another source can add at most one more operation.

Sequence numbers are needed for a source to track the operations of other sources. The FIFO property of the communication channels ensure that only one operation per source is queued at a target. This however does not guarantee that entries in different columns of the view are all part of a single operation, simply because they are from the same source. Due to message delays, one column can contain a more recent operation than another.

There is however a limit on how much columns can be skewed relative to each other. Sources only maintain a view when they have outstanding operations of their own. So we can guarantee that each column contains at least this one barrier operation. Then there are only two options for other operations: they arrived either before or after this barrier operation. Two sequence numbers are enough to distinguish them.

6 Asynchronous virtual shared disk model

The asynchronous model, described in the previous section, can be verified using the Spin model checker. For this purpose it needs to be implemented in Promela: the language used by Spin. First we will look at the Promela language constructs so we can continue with the implementation of a model for the asynchronous algorithm.

6.1 Promela

Promela, short for Process Meta Language, is a language for modeling parallel systems. It models the system as a collection of concurrent processes communicating through channels. Variables can be defined either globally or locally for processes. Control flow is handled using selection, repetition and break statements in combination with guards. Communication channels can be synchronous or asynchronous. Given a Promela model, the Spin model checker can verify its properties like correctness and liveliness.

Guards are an important part of the Promela language for handling nondeterminism. When a statement is not executable a process will block. This is true for example when trying to receive a message from an empty channel or comparing two unequal values. In combination with a selection statement, guards function as conditional statements:

```
if
:: a > b -> c
:: a < b -> d
:: else -> e
fi
```

The selection statement only chooses from executable statements. In other words, it chooses guard statements that will not block the execution of the process. The else guard is a special statement that is only executable when the other guards are not, which in this example is true when $a = b$.

A similar construct exists for repetitions, which can be exited from using a break statement:

```
do
:: a > b -> c
:: a < b -> d
:: else -> break
od
```

The repetition statement again only chooses from executable statements. The difference is that statements in the repetition construct can be executed repeatedly as long as the guards are executable. The break statement can be used to exit from the loop.

Channels are synchronous or asynchronous depending on their buffer size. A buffer size of 0 indicates a synchronous channel with rendezvous style com-

munication. A buffer size of $n > 0$ indicates an asynchronous channel which can store up to n messages:

```
chan x = [0] of { short };
chan y = [4] of { short };
```

Channel x is a synchronous channel for communicating values of the short datatype using rendezvous. Channel y is an asynchronous channel capable of storing up to four messages of the short datatype.

Values are sent and received on a channel using respectively exclamation and question marks in combination with the channel names:

```
x!32;
y?m;
```

The first statement sends the value 32 on channel x. The second statement receives a value from channel y and stores it in variable m. Messages in the channel are handled in FIFO order.

Promela does not allow the creation of functions; only inline macros are supported. An inline macro call in the code is directly substituted by its definition. As a consequence, no local variables can be defined for these inline definitions. Only global variables, accessible by all processes, and local variables, accessible by only one process, are available. This can make it hard to manage variables for larger models.

To reduce the state space searched by the SPIN model checker, it is important to clean up variables when they are no longer needed. Unused variables can differentiate program states that could otherwise be merged. Another way to reduce state space is to define deterministic steps:

```
d_step {
   /* some deterministic code */
}
```

Pieces of code that are not subject to non-determinism, for example from other processes, can be modeled as a single state transition. This significantly reduces the state space of the model. Because there are no states internal to the deterministic code, it is impossible to jump in or out of it. Possible non-determinism like choosing between multiple executable guards, is handled deterministically. It is an error if the code inside the deterministic step blocks.

6.2 Model

The following is a Promela model for concurrent operations on redundantly distributed data. Clients accessing the data are called sources: they initiate read and write operations. Servers storing the shared data are called targets: they service the read and write operations in some sequential order. The systems can consist of any number of sources and targets.

A source initiates an operation by communicating it to all targets, which store it in their queue. Due to concurrency, the order in which operations are placed in the queue can differ between targets. All targets should however commit the read and write requests in the same order; a coordinated decision needs to be made on which operations commit first.

The content of a target's queue is used as a repsonse to sources queuing an operation. Sources build up a combined view of all the target queues. This allows sources to determine the correct order to commit operations in and send this order to the targets. Since the decision algorithm ensures all sources reach the same conclusion, the operations will commit in the same order on all targets, despite possible concurrency.

6.2.1 Source queue

Sources run in a loop, sending out read and write operations. In practice these operations would be issued by other applications running on the source machine, but in this model we can simulate a constant stream of requests. The model checker will verify all possible interleavings of these requests, since message delivery can be arbitrarily delayed. Without loss off generality only write operations are considered: if writes are committed in identical order on all targets, then so would reads. Operations are first queued and then committed by the source:

```
do
:: s < SEQUENCE_NUMBERS ->
   /* receive write operation -> */
   source_queue(my_id + s * SOURCES);
   source_commit(my_id + s * SOURCES);
   d_step {
      util_wipe_view(queue_view);
      s++
   }
:: else -> s = 0
od
```

The acknowledgments from queuing the operation are stored in a queue view. This view is used by the commit algorithm to decide on a suitable order of operations. After the commit, when the write has finished, the view is cleaned for the next iteration.

Sequence numbers are used to distinguish between subsequent requests. It turns out that two sequence numbers are enough: once a source has queued a write, the other writes in the view are either from a previous or future commit. The sequence number combined with the identity of the source forms a unique id for each operation; ensuring no two different requests using the same id can be present in the queue view. For simplicity, writes are identified by this id, instead of the data read or written. Any data fields can be omitted from the protocol as a consequence.

A new write operation from the source is first queued. A message list is constructed containing only the id of this new operation:

```
d_step {
    assert(c == 0);

    /* queue the writes at the targets */
    message_list.el[0] = write;
    do
    :: c < TARGETS ->
        source_send(queue, c + 1, message_list);
        c++
    :: else -> c = 0; break
    od;
    util_wipe_queue(message_list)
}
```

All targets, numbered from 1 up to *TARGETS*, are sent a queue request with the constructed list. After cleaning the message list, the source will move on and wait for acknowledgments.

6.2.2 Target queue

Targets wait for incoming messages to process. Messages can be either queue or commit requests: queue message lists contain a single operation id, while commit message lists contain ids in the order in which they need to be committed:

```
progress: end: do
:: target_receive(type, id, message_list);
    d_step {
        if
        :: type == queue -> target_queue(id, message_list)
        :: type == commit -> target_commit(id, message_list)
        fi;
        type = 0; id = 0; util_wipe_queue(message_list)
    }
od
```

The message list, type variable and id variable, are cleaned after the request has been finished.

Queuing of operations on the target is relatively simple. All sources with requests already in the queue need to be informed of the new operation. They might need this new information to reach a decision. Since no source can have more than one operation in the queue, we can iterate through the queue list and send the source of each request a queue list update:

```
assert(q == 0);
do
:: q < QUEUE_SIZE && queue_list.el[q] != 0 ->
    target_send(queue_ack, ID(queue_list.el[q]), message_list);
    q++
:: else -> q = 0; break
od;

util_add_queue(message_list, queue_list);
target_send(queue_ack, id, queue_list)
```

The source of the new requests expects the complete queue in the acknowledgment, so it is important to add the new request after updates have been sent out. Only after the new request has been queued, is the acknowledgment sent back to the source.

6.2.3 Source commit

The source is waiting for queue acknowledgments of its operation. It expects to receive at least one acknowledgment from each target:

```
do
:: c < TARGETS ->
   source_receive(type, id, message_list);
   d_step {
      if
      :: type == commit_ack ->
         util_rem_queue(message_list, queue_view[id - 1]);
      :: type == queue_ack ->
         do
         :: q < QUEUE_SIZE ->
            if
            :: message_list.el[q] == write -> q = 0; c++; break
            :: else -> q++
            fi
         :: else -> q = 0; break
         od;
         util_add_queue(message_list, queue_view[id - 1]);
      fi;
      type = 0; id = 0; util_wipe_queue(message_list)
   }
:: else -> c = 0; break
od
```

Waiting for acknowledgments, it is possible to receive messages of other concurrent updates to the queue. These can be either commit acknowledgments, when another source is committing operations, or additional queue acknowledgments, when another source is also queuing operations.

Commit acknowledgments are directly processed by updating the queue view. Queue acknowledgments are not, since they can either be acknowledgments of the initial queue request or concurrent updates of the queue. We can distinguish between the two, because only the acknowledgments contain the id of the queue request in their message list. To ensure all targets acknowledged the initial queue request, the counter is only updated on queue request acknowledgments.

After all targets have acknowledged the queue request, the source will try to immediately commit the queued operations. This might fail when there is insufficient information for a decision. The source will then wait for additional messages, committing when possible and stopping only when its own operation has been committed on all targets:

```
do
```

```
:: source_receive(type, id, message_list);
   if
   :: type == commit_ack ->
      d_step {
         update_base(message_list);
         util_rem_queue(message_list, queue_view[id - 1]);
         type = 0; id = 0; util_wipe_queue(message_list);
      }
      break_when_done(write, 0);
      d_step {
         try_commit(write, base)
      }
   :: type == queue_ack ->
      d_step {
         util_add_queue(message_list, queue_view[id - 1]);
         type = 0; id = 0; util_wipe_queue(message_list);
         try_commit(write, base)
      }
   fi
od
```

The messages can be either commit or queue acknowledgments. Queue acknowledgments add ids to the queue view, while commit acknowledgments remove ids from the queue view. Both cases change the information in the queue view and give the source a new opportunity to try and commit.

The *base* variable indicates which rows of the queue view have already been committed. It prevents a source from repeatedly committing the same operations and is automatically incremented when the source successfully commits. The *base* variable is decremented when the first acknowledgment of a commit arrives.

6.2.4 Target commit

When commit requests are processed by the target, it is first checked if the operations are still in the queue. It is possible another source has concurrently committed the same operations and its request arrived earlier. The variable q is used to indicate success when set to $QUEUE\_SIZE$ and failure when set to 0:

```
do
:: q < QUEUE_SIZE && message_list.el[q] != 0 ->
   if
   :: count_list.el[message_list.el[q] - 1] > 0 -> q++
   :: else -> q = 0; break
   fi
:: else -> q = QUEUE_SIZE; break
od
```

A count list is used to check if an operation is in the queue. It has been created earlier using a utility function and contains the number of occurrences of operation id n at position $n-1$. The commit request is processed only when all operations in the message list are present in the queue, at which point the sources of all writes in the queue are informed about the commit:

```
do
:: c < COUNT_SIZE && count_list.el[c] != 0 ->
   target_send(commit_ack, c + 1, message_list);
   c++
:: c < COUNT_SIZE && count_list.el[c] == 0 ->
   c++
:: else -> c = 0; break
od;

util_rem_queue(message_list, queue_list)
```

This includes the original source of the commit request.

6.2.5 Commit algorithm

The heart of the model is formed by the commit algorithm, which tries to commit operations stored in the queue view. Whether this succeeds depends on the amount of information available and the progress of concurrent sources. First a conflict group needs to be found: a minimal number of consecutive rows in the queue view, containing exactly 0 or *TARGETS* occurrences of operations. It indicates a queue state where all targets have received the same operations, but possibly in a different order. The following code is repeated by an outer loop to find a conflict group, and will add a new row to the count list at each iteration:

```
util_count_row(q + base, queue_view, count_list);
do
:: c < COUNT_SIZE && count_list.el[c] == 0 -> c++
:: c < COUNT_SIZE && count_list.el[c] == TARGETS -> c++
:: else ->
   if
   :: c < COUNT_SIZE -> q++; c = 0; break
   :: else -> q++; break
   fi
od
```

The variable q is incremented each iteration and indicates the highest of the consecutive rows. The variable c is used to iterate over the different target queues in the the queue view. After the inner loop ends it doubles as an indicator for failure or success: failure when $c = 0$ and success when $c = COUNT\_SIZE$. On success a message list is created by concatenating the operations in the count list. The message list will be automatically ordered by operation id and can be sent to all targets:

```
util_create_message(message_list, count_list);
do
:: t < TARGETS ->
   source_send(commit, t + 1, message_list);
   t++
:: else -> t = 0; break
od
```

This algorithm in combination with the alternating sequence numbers, ensure that all sources decide on an identical commit order.

7 Asynchronous virtual shared disk implementation

7.1 MINIX

Contrary to monolithic operating systems like Windows and Linux, MINIX is designed to be modular. The MINIX kernel is kept small, while drivers and other services run as separate user-mode processes. Isolating these processes from the kernel improves reliability, since user-mode processes are less likely to take down the entire system. Services are implemented using server processes, like the file server, process server and reincarnation server. Drivers are implemented using device processes like the disk driver, network driver and audio driver. MINIX ensures these usermode processes can not execute any privileged or otherwise unsafe instructions: they have to use kernel calls.

The asynchronous implementation presented here is a normal user process. Implementing the algorithm as device process would have been preferable, but causes a layering violation. In MINIX the block and network drivers are accessed through the file system. An application accessing a virtual shared disk driver will also use the file server, blocking it in the process. This leaves the virtual shared disk driver unable to access either block or network driver. Even if we somehow bypass the blocked file server, these drivers will not be accessible by default. The network driver for example will only respond to requests from the file server. Without modifying MINIX internals, the only option is to run the algorithm as a user process.

Still, in preparation for more invasive changes to MINIX, the network is accessed by the implementation using native MINIX system calls. Using POSIX calls would not be appropriate for an operating system process.

7.2 Implementation

The following is a MINIX implementation of the Promela model in C. It mirrors the Promela model closely, except for some implementation specific adaptations. Contrary to the explanation of the Promela model, which follows the sequence of events leading to a successful write, this section will look at the separate implementations of the source and target code.

7.2.1 Source

The source is started on the command line using a parameter list of addresses. The list contains the addresses of all targets used by the protocol: each target identified using an ip and port pair. Targets can run on the same machine using different ports. The source starts by allocating memory:

```
queue = malloc(sizeof(action_t) * CONC_CONN * target_count);
if (queue == NULL) error("MALLOC");

queue_count = malloc(sizeof(int) * target_count);
```

```
if (queue_count == NULL) error("MALLOC");

for (i = 0; i < target_count; i++) {
  queue_count[i] = 0;
}
```

The *queue* array is used to store the queue view; each queue contains an array of *CONC_CONN* actions. The amount of queues needed is indicated by *target_count*: the number of target addresses passed when starting the source code. The *queue_count* stores the size of each queue.

A write operation is queued by sending it to all targets. Each message containing a single action comprised of the source address, a sequence number and the data to be written:

```
msg.type = QUEUE_REQ;
msg.size = 1;
msg.act = &act;

act.addr.ip = 0;
act.addr.port = 0;
act.seq = seq;
act.data = data;

for (i = 0; i < target_count; i++) {
  send(&target[i], &msg);
}

for (i = 0; i < target_count; i++) {
  doConfirm(&act);
}
```

The source waits for confirmation of the write action from all targets. The address and port number 0 have a special meaning: they are automatically replaced with the real ip and port of the source by the communication layer. This way, only the communication layer needs to know the ip and port numbers.

Care needs to be taken when counting confirmations. Additional update messages can be sent from a previously confirmed target. Such messages should be processed, but only messages containing the pending write should be counted:

```
do {
  recv(&src, &msg);

  pos = targetPosition(&src);
  if (pos < 0) error("POSITION");

  switch (msg.type) {
  case QUEUE_ACK :
    addQueue(msg.act, msg.size, queue[pos], &queue_count[pos]);
    confirmed = inQueue(pending, 1, msg.act, msg.size);
    break;
  case COMMIT_ACK :
    remQueue(msg.act, msg.size, queue[pos], &queue_count[pos]);
    confirmed = 0;
    break;
```

```
  }
} while (!confirmed);
```

Since the loop only exits on initial queue acknowledgments, only the correct acknowledgments are counted.

When the write action has been confirmed by all targets, the source moves on and tries to commit:

```
do {
  recv(&src, &msg);

  pos = targetPosition(&src);
  if (pos < 0) error("POSITION");

  switch (msg.type) {
  case QUEUE_ACK :
    addQueue(msg.act, msg.size, queue[pos], &queue_count[pos]);
    finished = 0;
    if (!pending) pending = tryCommit();
    break;
  case COMMIT_ACK :
    remQueue(msg.act, msg.size, queue[pos], &queue_count[pos]);
    finished = !needCommit(seq, data);
    if (!finished && !pending) pending = tryCommit();
    break;
  }
} while (!finished);
```

The *pending* variable indicates a previous commit is in progress; this prevents duplicate attempts to commit the same writes. The *finished* variable is set when no queue contains the initial write request anymore: all targets have committed the request and the source is done.

Determining whether a commit can be made is a lot easier in C than it is in Promela. A conflict group is found by comparing queue contents for increasing row numbers. When all queues contain the same actions, the row number indicates the last row of the conflict group:

```
for (i = 0; i < CONC_CONN; i++) {
  for (j = 0; j < target_count; j++) {
    n = (j + 1) % target_count;
    match = inQueue(queue[j], queue_count[j], queue[n], queue_count[n]);
    if (!match) break;
  }
  if (match) {
    sendCommit(i);
    return 1;
  }
}
```

Variable i iterates over the rows, while variable j iterates over the target queues. The first i actions in queue j are compared to the next queue n. There is a conflict group when all actions in the first i position of all the queues are identical.

7.2.2 Target

The target is waiting in an endless loop for incoming messages to process:

```
while(1) {
  recv(&src, &msg);

  switch (msg.type) {
    case QUEUE_REQ :
      doQueue(&src, &msg);
      break;
    case COMMIT_REQ :
      doCommit(&src, &msg);
      break;
    default :
      error("TYPE");
  }
}
```

Depending on the message type, the actions in the request are either queued or committed.

When the message is a request to queue, the incoming message can be easily transformed into an acknowledgment:

```
msg->type = QUEUE_ACK;

for (i = 0; i < queue_count; i++) {
  send(&queue[i].addr, msg);
}

addQueue(msg->act, msg->size, queue, &queue_count);

msg->size = queue_count;
msg->act = queue;

send(src, msg);
```

All other sources present in the queue are informed of the new action. The source of the original request is sent an acknowledgment containing the complete queue.

When the message is a request to commit, a check needs to be made for previous concurrent commits. Only when all actions are still present in the queue, is the commit executed:

```
queued = inQueue(msg->act, msg->size, queue, queue_count);
if (!queued) return;

for (i = 0; i < msg->size; i++) {
  value = msg->act[i].data;
}

msg->type = COMMIT_ACK;
for (i = 0; i < queue_count; i++) {
  send(&queue[i].addr, msg);
}
```

```
remQueue(msg->act, msg->size, queue, &queue_count);
```

The original message is reused for the acknowledgment. The sources of all actions in the queue need to be informed, including the source of the request.

8 Redundant virtual shared disk algorithm

The algorithm used in the previous sections has an important drawback: it does not allow for faulty processes. The asynchronous algorithm requires up-to-date information from all processes to decide on a correct ordering of operations. Therefore, a single crashing or otherwise unresponsive process can prevent read and write operations from completing.

This limitation does not exist for atomic registers. Atomic registers linearize read and write operations and can be constructed while only communicating with a majority of processes. Since only a majority of processes is needed, no minority of faulty processes can prevent the system from making progress. The system is wait-free when a majority of processes is correct. It might be possible to simulate a shared disk based on algorithms for atomic registers. There is however an issue with crash consistency.

8.1 Strict linearizability

Atomic register algorithms in message passing systems generally work on the basis of broadcasting messages to all processes in the system and waiting for responses from a majority of them. Writing to and reading from a majority of processes, ensures a newly written value is received by subsequent reads. The use of timestamps then enables the reader to select the most recent value from the responses. What happens however when a new value is written to only a minority of processes, because the writer crashes?

Partially written values can take effect at an arbitrary time in the future. Since the value has only been written to a minority of processes, there is no guarantee it will be received by subsequent reads. Assuming no new writes take place, this value can then surface in the system long after the original writer has crashed. One would prefer that subsequent reads are indicative of whether a preceding write failed or succeeded: similar to a shared disk.

The crash consistency problem described above, does not break the linearizability property of atomic registers. As far as atomic registers are concerned, partial writes never end. It is therefore correct, in terms of linearizability, for the written value to take effect at an arbitrary time in the future. To prevent this from happening, a new consistency model which includes crashing operations is required: strict linearizability [2].

A successful operation should take effect between its invocation and response, while a partial operation should take effect between its invocation and crash, or never take effect at all. These requirements for strict linearizability turn out to be impossible to implement in a wait-free manner for even simple atomic registers[2].

8.2 Probabilistic Consensus

Some form of consensus is needed to determine whether an operation will take effect or not. This ensures writes either take effect before a potential crash

or never at all. However, since partial operations can not be distinguished from slow operations there is a possibility for otherwise successful operations to be invalidated by consensus. Processes can no longer guarantee that every operation succeeds. Atomic registers for example would need to be extended with the ability for operations to abort.

Since deterministic consensus is impossible in asynchronous message passing systems, probabilistic consensus can be used instead [13, 11, 14]. A processes will try to reach consensus before finishing its operation. Because probabilistic consensus is not guaranteed, the operation will abort if consensus is not reached within a limited number of rounds. It is up to the user to determine what is an acceptable number of rounds. This maximum ensures the algorithm stays wait-free.

8.3 Timestamps

Atomic registers can use timestamps to distinguish between writes and determine which is the most recent value. However, using a probabilistic consensus algorithm based on rounds does not require explicit timestamps. Values can instead be identified with their respective round numbers: there can only be consensus on a single value in any particular round. The rounds of the consensus algorithm are in effect the timestamps.

Although not optimal, the round numbers used by the consensus algorithm will be unbounded. It is assumed that the set of numbers can be made large enough to never run out in any particular use case. Another option would be to use round numbers based on bounded timestamps [27, 16], but these exhibit a similar problem to the one we are trying to solve.

New bounded timestamps are calculated based on currently active timestamps. This ensures new timestamps are higher then any timestamps in the system. Active timestamps will therefore have to be recorded at a majority of processes. Recording timestamps however, is remarkably similar to writing values in atomic registers. Partially recorded timestamps can resurface at inopportune moments, potentially blocking the bounded timestamp algorithm.

8.4 Redundant algorithm

The redundant algorithm uses probabilistic consensus to strictly linearize virtual shared disk operations. Since consensus can not be guaranteed, operations have a chance to abort. Notice however that a consensus algorithm will always succeed if only a single new value is proposed. In other words: aborts can be ruled out when the virtual shared disk encounters no writes concurrent with other read or write operations. This is not an unreasonable assumption to make when the shared disk will be used to store a filesystem.

Deviating from this concurrency restriction does not compromise the integrity of the storage system. Though it is no longer guaranteed that the system will successfully complete the operation, it will not leave the system in an inconsistent state. Chances of an abort occurring can be reduced by increasing

the number of rounds allowed to reach consensus. This is a trade off between concurrency and responsiveness of the system.

Similar to the asynchronous version, the redundant algorithm uses multiple servers to store values and metadata. The metadata consists of the most current round and the last concurrent round encountered by the server. This information is needed to reach consensus: we can decide on a value when it is two rounds ahead of the last concurrent round. Other sequential or concurrent processes are guaranteed to detect such a value.

The algorithm behaves very similar for both reads and writes. The basic difference being that writes can introduce new values, while reads can only propagate existing values. Both operations start by the client requesting information on the current round from each server. When a majority of servers have responded, both operations proceed by rewriting the value of the highest round received. If multiple values are present in the highest round, the operations can choose any one of them. The rewrite ensures all subsequent operations will detect at least this highest round.

The subsequent round is started by either writing an existing value to the next round in case of a read operation or writing a new value in case of a write operation. The servers respond to each write by returning the value and metadata of the highest rounds they encountered. Read operations will adopt a new value if they detect one in a higher round. Write operations will retry if the value in a higher round is not their own. Both operations will abort if no consensus is reached within a predetermined number of rounds.

Consensus is reached when a value is two rounds ahead of the last concurrent round recorded by a majority of processes. This ensures that other concurrent operations will detect the new value and either adopt it or abort. One round is not sufficient, because a concurrent operation could still write to the current round. A two round lead can be reduced to a one round lead because of concurrency, but this is still sufficient for any concurrent process to detect the new value.

Servers store the value and metadata of the highest rounds they encountered. The value and current round number on the server is updated when receiving a value with a higher round number. The concurrent round number on the server is updated when receiving a value, different from the one currently stored. It is updated to the current round number of either the received value or the stored value, whichever is higher. In a sense the concurrent round number stores the last round where the current value was not the only value left. This makes it possible for clients to detect the lead of stored values for reaching consensus.

9 Redundant virtual shared disk model and implementation

In this section we will look at a combined model and implementation of a redundant agorithm for virtual shared disks. Contrary to chapters 6 and 7, code is shared between the Promela model and MINIX implementation. This allows for more flexibility in the Promela model, since C-code supports more language constructs; functions with local variables for example. The MINIX implemenation has the added benefit of being code pieces being model checked, ruling out any conceptual mistakes in those parts.

9.1 Server code

The server runs in a simple loop, receiving incoming request, processing them and sending out a response. Receiving and sending of messages is implemented in either Promela or C. The Spin model checker requires the use of Promela channels, while the MINIX implementation requires operating system specific C-code for communicating over ethernet. Both however share the use of the merge function, which can be seen in the following code snippet:

```
inline process() {
   receive_msg();

   c_code { merge(&Pserver->in, &Pserver->out); };

   send_msg()
}
```

The process macro function runs in an endless loop for each server process in the Promela model. Every time a message is received, the state of the server is updated by merging the new information (*in*) with the information already stored on the server (*out*). As a response, the updated information in *out* is sent to the client originating the initial request.

The merge C-code ensures stored data is correctly updated with new information. The data stored on the server consists of <*value*, *current*, *concurrent*> tuples. The value of the highest round encountered is stored by the server in *value*. The round this value was encountered in, is stored in *current*, while the last round where multiple values where active is stored in *concurrent*. These three values allow a client to determine when a value is two rounds ahead. They are updated as follows:

```
void merge(store_t *in, store_t *out) {
   if (in->value != out->value) {
      if (in->current > out->current) {
         out->concurrent = MAX(in->concurrent, out->current);
      } else {
         out->concurrent = MAX(in->current, out->concurrent);
      }
   } else {
```

```
        out->concurrent = MAX(in->concurrent, out->concurrent);
    }

    if (in->current > out->current) {
        out->value = in->value;
        out->current = in->current;
    }
}
```

First we look at the value of the incoming and stored tuples. Tuples storing values that are not equal can indicate concurrency: when adopting the tuple with the highest round, we need to ensure its concurrent round is at least as high as the current round of the other tuple. Tuples carrying equal values will never cause concurrency in their current voting rounds: the highest concurrent round can be determined solely by comparing earlier recorded concurrency. The last step is to adopt the value and current round of the tuple with the highest round.

9.2 Client code

Clients attempt to read and writes values in a limited number of steps. Operations can fail if probabilistic consensus is not reached within those steps: operations can abort. Similar to how tuples from clients are used to update tuples stored on servers, tuples from servers are used to update tuples on clients. Clients scatter tuples to every possible server and gather response tuples from at least a majority. The tuples are constructed by the client specifically to reach consensus for a desired operation. Take for example a read operation:

```
inline read() {
    assert(count_steps == 0);

    do
    :: count_steps < MAX_STEPS ->
        scatter_msg();
        gather_msg();
        count_steps++;

        if
        :: c_expr { read_done(&Pclient->store, &Pclient->out) } -> break
        :: else -> c_code { read_step(&Pclient->store, &Pclient->out); }
        fi

    :: else -> break
    od;

    count_steps = 0
}
```

The algorithm updates its local tuple by scattering and gathering messages. Shared C-code is used each step to determine if consensus has been reached, or to construct a new outgoing tuple if this is not the case. The goal is to have a value two rounds ahead of the competition: this guarantees concurrent processes

will detect the winning value when it is at least one round ahead. There are however restrictions to introducing a value into a new round.

Processes start with a tuple that is zeroed out. Such a tuple will have no influence on the servers, except for when the virtual shared disk has just been initialized. One could think of a freshly started virtual shared disk as being zeroed out; but even then, there is no consensus yet because of the zeroes in the concurrent fields. Responses to this tuple give the client a starting point for its operation.

Before a new round can be initiated, it needs to be certain the previous round has reached a majority of servers. This prevents processes from repeatedly advancing a round and crashing; would this be the case, then consensus would no longer be guaranteed for values two rounds ahead. Therefore the algorithm will first rewrite the highest round it encountered to a majority of servers. Only then will it advance to a higher round.

The algorithm keeps track of these different states by separately storing the outgoing tuple from the merged tuple. After scattering and gathering, these tuples can be compared. New rounds can be started when the merged tuple has the same current round as the outgoing tuple. If the merged tuple has a higher current round, then a rewrite is needed for this round. The actual comparison is done in C-code:

```
void read_step(store_t *store, store_t *out) {
    out->value = store->value;
    out->concurrent = store->concurrent;

    if (store->current == out->current) {
        out->current = store->current + 1;
    } else {
        out->current = store->current;
    }
}
```

The merged tuple in *store* is guaranteed to incorporate the tuple in *out* after a scatter and gather operation; the value and concurrent round can be safely copied. Depending on the current rounds of both tuples we either start a new round when they are equal, or rewrite the most recent round when they are not. This still leaves detecting consensus when a value is two rounds ahead of the competion. For this another piece of C-code is used:

```
int read_done(store_t *store, store_t *out) {
    if (store->current == store->concurrent + 2) {
        return store->current == out->current;
    }

    return store->current > store->concurrent + 2;
}
```

It uses the same reasoning as the previous function to determine consensus. It is not enough to detect a value that is two rounds ahead, we need to be sure this information has reached a majority. Separately storing the last outgoing

tuple can help guarantee a majority has been informed. Detecting a value more then two rounds ahead, guarantees a majority will have it stored at least two rounds ahead.

A write operation functions not much different than the read operation used as an example above. The only difference is that a write operation will stick to its own value. Compare the following C-code for taking a write step, with that of taking a read step:

```
void write_step(store_t *store, store_t *out) {
   out->concurrent = store->concurrent;

   if (store->current == out->current) {
      out->current = store->current + 1;
   } else {
      out->current = store->current;
   }
}
```

The only difference is in the missing first line compared to the read step. Read operations will adopt other values, if that will increase their chances to reach consensus. Write operations on the other hand will only want to force their value onto the system. A more explicit example can be seen in the following listing. It shows how a write operation will only finish when its own value has reached consensus:

```
int write_done(store_t *store, store_t *out) {
   if (store->value != out->value) {
      return FALSE;
   } else {
      return read_done(store, out);
   }
}
```

10 Conclusion

The promising ability of atomic registers to reach consistency without consensus, can not be directly translated to a virtual shared disk algorithm. Some form of consensus is still needed to prevent failing processes from breaking consistency when crashes are externally observable. While using probabilistic consensus does not guarantee termination in a limited number of steps, this risk can be mitigated.

It is not unreasonable to expect the primary application of a virtual shared disk to be the storage of filesystems. A shared disk file system will prevent unwanted concurrency on the underlying storage system, limiting it to at most concurrent reads. This paper shows how a reliable virtual shared disk can be implemented when adhering to these constraints. If concurrency is limited to concurrent reads, all operations are guaranteed to succeed.

Still, operations that break the concurrency constraints are not sure to fail. Neither do they endanger the integrity of the storage system. The inherit benefit of using probabilistic consensus is the chance to successfully complete any concurrent operation. Increasing the number of rounds allowed for operations in the virtual shared disk algorithm, will increase the chance of concurrent operations to succeed. It is up to the user to balance system responsiveness with concurrency requirements.

References

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distrib. Comput.*, 13(2):99–125, 2000.

[2] M.K. Aguilera and S Frolund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Laboratories Palo Alto, 2003.

[3] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and P.W. Hutto. Causal memory: Definitions, implementation and programming. Technical Report GIT-CC-93-55, Georgia Institute of Technology, 1993.

[4] James H. Anderson, Ambuj K. Singh, and Mohamed G. Gouda. The elusive atomic register. Technical report, Austin, TX, USA, 1986.

[5] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks and multi-processor coordination. In *STOC '91: Proceedings of the twenty-third annual ACM Symposium on Theory of Computing*, pages 348–358, New York, NY, USA, 1991. ACM.

[6] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[7] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 222–231, New York, NY, USA, 1987. ACM.

[8] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[9] Pei Cao, Swee Boon Lin, Shivakumar Venkataraman, and John Wilkes. The tickertaip parallel raid architecture. *ACM Trans. Comput. Syst.*, 12(3):236–269, 1994.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[11] Ling Cheung. Randomized wait-free consensus using an atomicity assumption. In *Proceedings OPODIS 2005*, pages 36–45, 2005.

[12] Benny Chor, Amos Israeli, and Ming Li. On processor coordination using asynchronous hardware. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 86–97, New York, NY, USA, 1987. ACM.

[13] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, 1994.

[14] Carole Delporte-Gallet and Hugues Fauconnier. Two consensus algorithms with atomic registers and failure detector omega. In *10th International Conference on Distributed Computing and Networking*, pages 251–262, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Danny Dolev, Idit Keidar, and Esti Yeger Lotem. Dynamic voting for consistent primary components. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 63–71, New York, NY, USA, 1997. ACM.

[16] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM J. Comput.*, 26(2):418–455, 1997.

[17] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 236–245, New York, NY, USA, 2004. ACM.

[18] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[19] Lars Ellenberg. Drbd 9 & device-mapper. In *Proceedings of Linux-Kongress 2008 October 7-10*, Hamburg, Germany, 2008.

[20] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, April 1985.

[21] Chryssis Georgiou, Nicolas Nicolaou, and Alexander A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 425–425, New York, NY, USA, 2008. ACM.

[22] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 281–290, New York, NY, USA, 2006. ACM.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[24] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst.*, 4(1):32–53, 1986.

[25] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[26] Hui i Hsiao and David J. Dewitt. Chained declustering: A new availability strategy for multiprocssor database machines. In *in Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.

[27] Amos Israeli and Ming Li. Bounded time-stamps. *Distrib. Comput.*, 6(4):205–209, 1993.

[28] Amos Israeli and Amnon Shaham. Optimal multi-writer multi-reader atomic register. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 71–82. ACM Press, 1992.

[29] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[30] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[31] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[32] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[33] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 84–92, New York, NY, USA, 1996. ACM.

[34] Ming Li, John Tromp, and Paul M. B. Vitányi. How to share concurrent wait-free variables. *J. ACM*, 43(4):723–746, 1996.

[35] Nancy Lynch and Alex A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *In DISC*, pages 173–190, 2002.

[36] Richard Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 232–248, New York, NY, USA, 1987. ACM.

[37] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[38] Gary L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

[39] Gary L. Peterson and James E. Burns. Concurrent reading while writing II: The multi-writer case. In *SFCS '87: Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 383–392, Washington, DC, USA, 1987. IEEE Computer Society.

[40] Philipp Reisner. Drbd v8: Replicated storage with shared disk semantics. In *Proceedings of the 12th International Linux System Technology Conference October 11-14*, University of Hamburg, Germany, 2005.

[41] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 48–58, New York, NY, USA, 2004. ACM.

[42] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

[43] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register revisited. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 206–221, New York, NY, USA, 1987. ACM.

[44] Robbert van Renesse and Andrew S. Tanenbaum. Voting with ghosts. In *Proc. 8th International Conference on Distributed Computing Systems*, pages 456–462, 1988.

[45] Paul M. B. Vitanyi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. *Symposium on Foundations of Computer Science*, pages 233–243, 1986.

[46] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–282, 1989.

A Promela code of the asynchronous algorithm

```
#define TARGETS 2
#define SOURCES 2
#define CHAN_SIZE 10
#define SEQUENCE_NUMBERS 2

#define CHANNELS 8 /* TARGETS * SOURCES * 2 */
#define QUEUE_SIZE 2 /* SOURCES */
#define COUNT_SIZE 4 /* SOURCES * SEQUENCE_NUMBERS */

#define ID(write) \
   (((write - 1) % SOURCES) + 1)
#define INDEX(direction, source, target) \
   (direction == forward -> \
   (source - 1) * TARGETS + (target - 1) : \
   (source - 1) * TARGETS + (target - 1) + TARGETS * SOURCES)
#define CHANNEL(direction, source, target) \
   channels[INDEX(direction, source, target)]

typedef queue_array {
   byte el[QUEUE_SIZE] = 0}
typedef count_array {
   byte el[COUNT_SIZE] = 0}

queue_array disk[TARGETS];

mtype = {forward, backward}
mtype = {queue, commit, queue_ack, commit_ack};
chan channels[CHANNELS] = [CHAN_SIZE] of {mtype, queue_array};

inline util_add_queue(a, b) { d_step {
   /* append elements from queue a to queue b */
   assert(i == 0);
   assert(j == 0);

   do
   :: i < QUEUE_SIZE && b.el[i] != 0 ->
      i++
   :: else -> break
   od;

   do
   :: j < QUEUE_SIZE && a.el[j] != 0 ->
      b.el[i] = a.el[j];
      i++; j++
   :: else -> i = 0; j = 0; break
   od;

   skip
} }

inline util_rem_queue(a, b) { d_step {
   /* remove elements in queue a from queue b */
   assert(i == 0);
   assert(j == 0);
```

```
    /* set elements in b equal to 0 if they are in a */
    do
    :: i < QUEUE_SIZE && a.el[i] != 0 ->
       do
       :: j < QUEUE_SIZE ->
          if
          :: a.el[i] == b.el[j] -> b.el[j] = 0
          :: else -> skip
          fi;
          j++
       :: else -> j = 0; break
       od;
       i++
    :: else -> i = 0; break
    od;

    /* compact b by removing holes caused by above */
    do
    :: i < QUEUE_SIZE && j < QUEUE_SIZE ->
       if
       :: b.el[i] != 0 -> i++; j++
       :: b.el[i] == 0 && b.el[j] == 0 -> j++
       :: else ->
          b.el[i] = b.el[j];
          b.el[j] = 0;
          i++; j++
       fi
    :: else -> i = 0; j = 0; break
    od;

    skip
} }

inline util_count(w, n, a, b) { d_step {
    /* count the ids or writes in the n queues of array a by
       increasing the corresponding values in count_array b */
    assert(i == 0);
    assert(j == 0);

    do
    :: i < n ->
       do
       :: j < QUEUE_SIZE && a[i].el[j] != 0 ->
          if
          :: w == 0 -> b.el[ID(a[i].el[j]) - 1]++
          :: w == 1 -> b.el[a[i].el[j] - 1]++
          fi;
          j++
       :: else -> j = 0; break
       od;
       i++
    :: else -> i = 0; break
    od;

    skip
} }
```

```
inline util_count_ids(a, b) {
   /* count the ids in queue a by increasing
      the corresponding values in count_array b */
   util_count(0, 1, a, b)
}

inline util_count_list(a, b) {
   /* count the writes in queue a by increasing
      the corresponding values in count_array b */
   util_count(1, 1, a, b)
}

inline util_count_view(a, b) {
   /* count the writes in view a by increasing
      the corresponding values in count_array b */
   util_count(1, TARGETS, a, b)
}

inline util_wipe_list(n, a) { d_step {
   /* reset the list of length n to all zeros */
   assert(i == 0);

   do
   :: i < n ->
      a.el[i] = 0;
      i++
   :: else -> i = 0; break
   od;

   skip
} }

inline util_wipe_queue(a) {
   util_wipe_list(QUEUE_SIZE, a)
}

inline util_wipe_count(a) {
   util_wipe_list(COUNT_SIZE, a)
}

inline util_wipe_view(v) { d_step {
   assert(i == 0);
   assert(j == 0);

   do
   :: i < TARGETS ->
      do
      :: j < QUEUE_SIZE ->
         v[i].el[j] = 0;
         j++
      :: else -> j = 0; break
      od;
      i++
   :: else -> i = 0; break
   od;

   skip
```

```
} }

inline util_count_row(n, v, b) { d_step {
   /* count the writes at position n in the arrays of
      view v and add those to count_array b */
   assert(i == 0);

   do
   :: i < TARGETS ->
      if
      :: v[i].el[n] != 0 -> b.el[v[i].el[n] - 1]++;
      :: else -> skip
      fi;
      i++
   :: else -> i = 0; break
   od;

   skip
} }

inline util_rem_row(v) { d_step {
   /* remove row 0 from view v */
   assert(i == 0);
   assert(j == 0);

   do
   :: i < TARGETS ->
      do
      :: j < QUEUE_SIZE - 1 ->
         v[i].el[j] = v[i].el[j + 1];
         j++
      :: else ->
         v[i].el[j] = 0;
         j = 0; break
      od;
      i++
   :: else -> i = 0; break
   od;

   skip
} }

inline util_create_message(m, b) { d_step {
   /* create message m adding all writes counted in b */
   assert(i == 0);
   assert(j == 0);

   do
   :: i < COUNT_SIZE ->
      if
      :: b.el[i] != 0 ->
         m.el[j] = i + 1;
         j++; i++
      :: else -> i++;
      fi
   :: else -> i = 0; j = 0; break
   od;
```

```
   skip
} }

inline disk_write(writes) {
   util_add_queue(writes, disk[my_id - 1]);

   assert(t == 0);
   do
   :: t < TARGETS ->
      if
      :: disk[t].el[0] != 0 ->
         assert(disk[0].el[0] == disk[t].el[0]);
         t++
      :: else -> t = 0; break
      fi
   :: else ->
      util_rem_row(disk);
      t = 0
   od
}

inline target_send(type, id, message_list) {
   CHANNEL(backward, id, my_id)!type(message_list)
}

inline target_receive(type, id, message_list) {
   if
   :: d_step {
         CHANNEL(forward, 1, my_id)?type(message_list) ->
         id = 1
      }
   :: d_step {
         SOURCES > 1 && nempty(CHANNEL(forward, 2, my_id)) ->
         CHANNEL(forward, 2, my_id)?type(message_list);
         id = 2
      }
   fi
}

inline target_queue(id, message_list) {
   assert(q == 0);
   do
   :: q < QUEUE_SIZE && queue_list.el[q] != 0 ->
      target_send(queue_ack, ID(queue_list.el[q]), message_list);
      q++
   :: else -> q = 0; break
   od;

   util_add_queue(message_list, queue_list);
   target_send(queue_ack, id, queue_list)
}

inline target_commit(id, message_list) {
   util_count_list(queue_list, count_list);

   assert(q == 0);
```

```
    do /* check if writes are still in queue */
    :: q < QUEUE_SIZE && message_list.el[q] != 0 ->
       if
       :: count_list.el[message_list.el[q] - 1] > 0 -> q++
       :: else -> q = 0; break
       fi
    :: else -> q = QUEUE_SIZE; break
    od;

    util_wipe_count(count_list);
    util_count_ids(queue_list, count_list);

    if /* id and writes have not already been commited */
    :: q > 0 && count_list.el[id - 1] != 0 ->

       /* commit writes to disk */
       disk_write(message_list);

       assert(c == 0);
       do /* inform ids in queue about the commit */
       :: c < COUNT_SIZE && count_list.el[c] != 0 ->
          target_send(commit_ack, c + 1, message_list);
          c++
       :: c < COUNT_SIZE && count_list.el[c] == 0 ->
          c++
       :: else -> c = 0; break
       od;

       q = 0; util_rem_queue(message_list, queue_list)
    :: else -> q = 0; skip
    fi;

    util_wipe_count(count_list)
}

proctype Target(byte my_id) {
    queue_array queue_list, message_list;
    mtype type;
    count_array count_list;
    byte id, q, c, t;

    byte i, j;

    progress: end: do
    :: target_receive(type, id, message_list);
       d_step {
          if
          :: type == queue -> target_queue(id, message_list)
          :: type == commit -> target_commit(id, message_list)
          fi;
          type = 0; id = 0; util_wipe_queue(message_list)
       }
    od
}

inline source_send(type, id, message_list) {
    CHANNEL(forward, my_id, id)!type(message_list)
```

```
}

inline source_receive(type, id, message_list) {
   if
   :: d_step {
      CHANNEL(backward, my_id, 1)?type(message_list) ->
      id = 1
   }
   :: d_step {
      TARGETS > 1 && nempty(CHANNEL(backward, my_id, 2)) ->
      CHANNEL(backward, my_id, 2)?type(message_list);
      id = 2
   }
   fi
}

inline source_queue(write) {
   d_step {
      assert(c == 0);

      /* queue the writes at the targets */
      message_list.el[0] = write;
      do
      :: c < TARGETS ->
         source_send(queue, c + 1, message_list);
         c++
      :: else -> c = 0; break
      od;
      util_wipe_queue(message_list)
   }

   do
   :: c < TARGETS ->
      source_receive(type, id, message_list);
      d_step {
         if
         :: type == commit_ack ->
            util_rem_queue(message_list, queue_view[id - 1]);
         :: type == queue_ack ->
            do
            :: q < QUEUE_SIZE ->
               if
               :: message_list.el[q] == write -> q = 0; c++; break
               :: else -> q++
               fi
            :: else -> q = 0; break
            od;
            util_add_queue(message_list, queue_view[id - 1]);
         fi;
         type = 0; id = 0; util_wipe_queue(message_list)
      }
   :: else -> c = 0; break
   od
}

inline update_base(message_list) {
   assert(q == 0);
```

```
    assert(t == 0);

    util_count_list(message_list, count_list);

    do
    :: q < QUEUE_SIZE && t == 0 ->
       do
       :: t < TARGETS && queue_view[t].el[q] != 0 ->
          if
          :: count_list.el[queue_view[t].el[q] - 1] != 0 -> t++
          :: else -> t++; break
          fi
       :: t < TARGETS && queue_view[t].el[q] == 0 -> t++; break
       :: else -> base--; q++; t = 0; break
       od
    :: else -> q = 0; t = 0; break
    od;

    util_wipe_count(count_list)
}

inline break_when_done(write, base) {
    d_step {
       assert(q == 0);
       assert(t == 0);

       q = base;

       do
       :: q < QUEUE_SIZE && t == 0 ->
          do
          :: t < TARGETS ->
             if
             :: queue_view[t].el[q] == write -> t++; break
             :: else -> t++
             fi
          :: else -> q++; t = 0; break
          od
       :: else -> q = 0; break
       od;

       skip
    }

    if
    :: t != 0 -> t = 0
    :: else -> break
    fi
}

inline try_commit(write, base) {
    assert(q == 0);
    assert(c == 0);

    do
    :: q + base < QUEUE_SIZE && c == 0 ->
       /* search for conflict group */
```

```
        util_count_row(q + base, queue_view, count_list);
        do
        :: c < COUNT_SIZE && count_list.el[c] == 0 -> c++
        :: c < COUNT_SIZE && count_list.el[c] == TARGETS -> c++
        :: else ->
           if
           :: c < COUNT_SIZE -> q++; c = 0; break
           :: else -> q++; break
           fi
        od

    :: c == COUNT_SIZE ->
        /* commit conflict group */

        c = 0;
        util_create_message(message_list, count_list);
        util_wipe_count(count_list);

        if
        :: message_list.el[0] == 0 -> q = 0; break
        :: else -> base = base + q; q = 0
        fi;

        assert(t == 0);
        do
        :: t < TARGETS ->
           source_send(commit, t + 1, message_list);
           t++
        :: else -> t = 0; break
        od;

        util_wipe_queue(message_list);

        break_when_done(write, base)

    :: else ->
        /* stop searching */

        q = 0;
        util_wipe_count(count_list);
        break

    od;
    skip
}

inline source_commit(write) {
    d_step {
        assert(base == 0);
        try_commit(write, base)
    }

    do
    :: source_receive(type, id, message_list);
        if
        :: type == commit_ack ->
```

```
            d_step {
               update_base(message_list);
               util_rem_queue(message_list, queue_view[id - 1]);
               type = 0; id = 0; util_wipe_queue(message_list);
            }
            break_when_done(write, 0);
            d_step {
               try_commit(write, base)
            }
         :: type == queue_ack ->
            d_step {
               util_add_queue(message_list, queue_view[id - 1]);
               type = 0; id = 0; util_wipe_queue(message_list);
               try_commit(write, base)
            }
         fi
      od;

      base = 0
   }

   proctype Source(byte my_id) {
      queue_array queue_view[TARGETS], message_list;
      mtype type;
      count_array count_list;
      byte id, base;
      byte s, t, c, q;

      byte i, j;

      do
      :: s < SEQUENCE_NUMBERS ->
         /* receive write operation -> */
         source_queue(my_id + s * SOURCES);
         source_commit(my_id + s * SOURCES);
         d_step {
            util_wipe_view(queue_view);
            s++
         }
      :: else -> s = 0
      od
   }

   init {
      byte t, s;

      atomic {
         do
         :: t < TARGETS ->
            run Target(t + 1);
            t++
         :: s < SOURCES ->
            run Source(s + 1);
            s++
         :: else -> break
         od
      }
```

```
}
```

B MINIX code of the asynchronous algorithm

B.1 channel.h

```
void send(address_t *dst, message_t *msg);
void recv(address_t *src, message_t *msg);
void print(message_t *msg);
void error(char *msg);

address_t *myAddress(void);
```

B.2 channel.c

```
#include "default.h"
#include "channel.h"

#define DEFAULT_DEVICE "/dev/tcp"

#define LISTEN_MSG "Listening on %s:%u.\n"

address_t my_address;

int listen_fd;
int listen_count = 0;

int open_fd[CONC_CONN];
int open_count = 0;

void report(int fd)
{
  int err;
  nwio_tcpconf_t nwio_tcpconf;

  err = ioctl(fd, NWIOGTCPCONF, &nwio_tcpconf);
  if (err) error("NWIOGTCPCONF");

  printf(LISTEN_MSG,
    inet_ntoa(nwio_tcpconf.nwtc_locaddr),
    ntohs(nwio_tcpconf.nwtc_locport));
}

int listenChannel(void)
{
  int fd, err;
  char *tcp_device;
  int backlog = CONC_CONN;

  nwio_tcpconf_t nwio_tcpconf;

  if ((tcp_device = getenv("TCP_DEVICE")) == NULL)
    tcp_device = DEFAULT_DEVICE;
  fd = open(tcp_device, O_RDWR);
  if (fd < 0) error("TCP_DEVICE");
```

```
  nwio_tcpconf.nwtc_flags = NWTC_COPY | NWTC_LP_SEL;
  nwio_tcpconf.nwtc_flags |= NWTC_UNSET_RA | NWTC_UNSET_RP;

  err = ioctl(fd, NWIOSTCPCONF, &nwio_tcpconf);
  if (err) error("NWIOSTCPCONF");
  err = ioctl(fd, NWIOTCPLISTENQ, &backlog);
  if (err) error("NWIOTCPLISTENQ");

  err = ioctl(fd, NWIOGTCPCONF, &nwio_tcpconf);
  if (err) error("NWIOGTCPCONF");

  my_address.ip = nwio_tcpconf.nwtc_locaddr;
  my_address.port = nwio_tcpconf.nwtc_locport;

  report(fd);
  return fd;
}

int acceptChannel(int listen)
{
  int fd, err;
  char *tcp_device;
  tcp_cookie_t cookie;

  if ((tcp_device = getenv("TCP_DEVICE")) == NULL)
    tcp_device = DEFAULT_DEVICE;
  fd = open(tcp_device, O_RDWR);
  if (fd < 0) error("TCP_DEVICE");

  err = ioctl(fd, NWIOGTCPCOOKIE, &cookie);
  if (err) error("NWIOGTCPCOOKIE");
  err = ioctl(listen, NWIOTCPACCEPTTO, &cookie);
  if (err) error("NWIOTCPACCEPTTO");

  return fd;
}

int openChannel(address_t *dst)
{
  int fd, err;
  char *tcp_device;

  nwio_tcpconf_t nwio_tcpconf;
  nwio_tcpcl_t nwio_tcpcl;

  if ((tcp_device = getenv("TCP_DEVICE")) == NULL)
    tcp_device = DEFAULT_DEVICE;
  fd = open(tcp_device, O_RDWR);
  if (fd < 0) error("TCP_DEVICE");

  if (my_address.ip == 0) {
    nwio_tcpconf.nwtc_flags = NWTC_LP_SEL;
  } else {
    nwio_tcpconf.nwtc_locport = my_address.port;
    nwio_tcpconf.nwtc_flags = NWTC_LP_SET;
  }
```

```
  nwio_tcpconf.nwtc_remaddr = dst->ip;
  nwio_tcpconf.nwtc_remport = dst->port;
  nwio_tcpconf.nwtc_flags |= NWTC_COPY | NWTC_SET_RA | NWTC_SET_RP;

  nwio_tcpcl.nwtcl_flags = 0;

  err = ioctl(fd, NWIOSTCPCONF, &nwio_tcpconf);
  if (err) error("NWIOSTCPCONF");
  err = ioctl(fd, NWIOTCPCONN, &nwio_tcpcl);
  if (err) error("NWIOTCPCONN");

  err = ioctl(fd, NWIOGTCPCONF, &nwio_tcpconf);
  if (err) error("NWIOGTCPCONF");

  my_address.ip = nwio_tcpconf.nwtc_locaddr;
  my_address.port = nwio_tcpconf.nwtc_locport;

  return fd;
}

void closeChannel(int fd)
{
  int i, err;
  int shift = 0;

  for (i = 0; i < open_count; i++) {
    if (open_fd[i] == fd) {
      err = ioctl(fd, NWIOTCPSHUTDOWN, NULL);
      if (err) error("NWIOTCPSHUTDOWN");
      close(fd);

      open_count--;
      shift++;
    }

    if (shift > 0) {
      open_fd[i] == open_fd[i + shift];
    }
  }
}

int selectFd(void)
{
  fd_set rfds;
  int i, count, max_fd;

  if (listen_count <= 0 && open_count <= 0) {
    listen_fd = listenChannel();
    listen_count = 1;
  }

  FD_ZERO(&rfds);
  max_fd = 0;

  if (listen_count > 0) {
    FD_SET(listen_fd, &rfds);
    if (listen_fd > max_fd) max_fd = listen_fd;
```

```
  }

  for (i = 0; i < open_count; i++) {
    FD_SET(open_fd[i], &rfds);
    if (open_fd[i] > max_fd) max_fd = open_fd[i];
  }

  count = select(max_fd + 1, &rfds, NULL, NULL, NULL);
  if (count <= 0) error("SELECT");

  if (FD_ISSET(listen_fd, &rfds)) {
    open_fd[open_count++] = acceptChannel(listen_fd);
    return open_fd[open_count - 1];
  }

  for (i = 0; i < open_count; i++) {
    if (FD_ISSET(open_fd[i], &rfds)) return open_fd[i];
  }

  error("SELECT");
}

int getFd(address_t *dst)
{
  int i, err;
  nwio_tcpconf_t nwio_tcpconf;

  for (i = 0; i < open_count; i++) {
    err = ioctl(open_fd[i], NWIOGTCPCONF, &nwio_tcpconf);
    if (err) error("NWIOGTCPCONF");

    if (dst->ip == nwio_tcpconf.nwtc_remaddr &&
    dst->port == nwio_tcpconf.nwtc_remport) {
      return open_fd[i];
    }
  }

  return open_fd[open_count++] = openChannel(dst);
}

ssize_t recvAction(int fd, action_t *act)
{
  ssize_t ssize = 0, done = 0;

  do {
    ssize = read(fd, (char *)act + done, sizeof(action_t) - done);
    if (ssize <= 0) return ssize;
    done += ssize;
  } while (done < sizeof(action_t));

  if (act->addr.ip == my_address.ip && act->addr.port == my_address.port) {
    act->addr.ip = 0;
    act->addr.port = 0;
  }

  return 1;
}
```

```
ssize_t recvMessage(int fd, message_t *msg)
{
  int i;
  ssize_t ssize;

  ssize = read(fd, &msg->type, sizeof(msg->type));
  if (ssize <= 0) return ssize;

  ssize = read(fd, &msg->size, sizeof(msg->size));
  if (ssize <= 0) return ssize;

  for (i = 0; i < msg->size; i++) {
    ssize = recvAction(fd, &msg->act[i]);
    if (ssize <= 0) return ssize;
  }

  return 1;
}

ssize_t sendAction(int fd, action_t *act)
{
  ssize_t ssize = 0, done = 0;

  if (act->addr.ip == 0 && act->addr.port == 0) {
    act->addr.ip = my_address.ip;
    act->addr.port = my_address.port;
  }

  do {
    ssize = write(fd, (char *)act + done, sizeof(action_t) - done);
    if (ssize <= 0) return ssize;
    done += ssize;
  } while (done < sizeof(action_t));

  if (act->addr.ip == my_address.ip && act->addr.port == my_address.port) {
    act->addr.ip = 0;
    act->addr.port = 0;
  }

  return 1;
}

ssize_t sendMessage(int fd, message_t *msg)
{
  int i;
  ssize_t ssize;

  ssize = write(fd, &msg->type, sizeof(msg->type));
  if (ssize <= 0) return ssize;

  ssize = write(fd, &msg->size, sizeof(msg->size));
  if (ssize <= 0) return ssize;

  for (i = 0; i < msg->size; i++) {
    ssize = sendAction(fd, &msg->act[i]);
    if (ssize <= 0) return ssize;
```

```
  }

  return 1;
}

void recv(address_t *src, message_t *msg)
{
  int fd;
  ssize_t ssize;
  nwio_tcpconf_t nwio_tcpconf;

  do {
    fd = selectFd();
    ssize = recvMessage(fd, msg);
    if (ssize == 0) closeChannel(fd);
  } while (ssize == 0);

  ioctl(fd, NWIOGTCPCONF, &nwio_tcpconf);
  src->ip = nwio_tcpconf.nwtc_remaddr;
  src->port = nwio_tcpconf.nwtc_remport;
}

void send(address_t *dst, message_t *msg)
{
  int fd;
  ssize_t ssize;

  fd = getFd(dst);
  ssize = sendMessage(fd, msg);
  if (ssize < 1) error("SEND");
}

address_t *myAddress(void)
{
  return &my_address;
}
```

B.3 default.h

```
#define _MINIX_SOURCE 1

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <stddef.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <net/hton.h>
#include <net/gen/in.h>
#include <net/gen/inet.h>
#include <net/gen/tcp.h>
#include <net/gen/tcp_io.h>
```

```
#define QUEUE_REQ 1
#define COMMIT_REQ 2
#define QUEUE_ACK 3
#define COMMIT_ACK 4

#define CONC_CONN 32

typedef struct {
  ipaddr_t ip;
  tcpport_t port;
} address_t;

typedef struct {
  address_t addr;
  u8_t seq;
  u8_t data;
} action_t;

typedef struct {
  u8_t type;
  u8_t size;
  action_t *act;
} message_t;

void error(char *msg);
```

B.4 default.c

```
#include "default.h"

void error(char *msg)
{
  printf("%s: %s\n", msg, strerror(errno));
  exit(-1);
}
```

B.5 source.c

```
#include "default.h"
#include "channel.h"
#include "util.h"

#define IP_STRING_SIZE 16
#define MAX_OCTET_NUMBER 255
#define MAX_PORT_NUMBER 65535

address_t target[CONC_CONN];
int target_count = 0;

action_t (*queue)[CONC_CONN];
int *queue_count;

int readAddress(char *arg, address_t *addr)
{
  int count;
  int octet[4], port;
```

```
  char ip[IP_STRING_SIZE];

  count = sscanf(arg, "%d.%d.%d.%d:%d",
         &octet[0], &octet[1], &octet[2], &octet[3], &port);
  if (count != 5) return -1;

  if (octet[0] > MAX_OCTET_NUMBER ||
      octet[1] > MAX_OCTET_NUMBER ||
      octet[2] > MAX_OCTET_NUMBER ||
      octet[3] > MAX_OCTET_NUMBER ||
      port > MAX_PORT_NUMBER) return -1;

  sprintf(ip, "%d.%d.%d.%d", octet[0], octet[1], octet[2], octet[3]);

  addr->ip = inet_addr(ip);
  addr->port = htons(port);

  return 0;
}

int targetPosition(address_t *addr)
{
  int i;

  for (i = 0; i < target_count; i++) {
    if (target[i].ip == addr->ip &&
        target[i].port == addr->port) return i;
  }

  return -1;
}

void doConfirm(action_t *pending)
{
  address_t src;
  message_t msg;
  action_t act[CONC_CONN];
  int confirmed, pos;

  msg.act = act;

  do {
    recv(&src, &msg);

    pos = targetPosition(&src);
    if (pos < 0) error("POSITION");

    switch (msg.type) {
    case QUEUE_ACK :
      addQueue(msg.act, msg.size, queue[pos], &queue_count[pos]);
      confirmed = inQueue(pending, 1, msg.act, msg.size);
      break;
    case COMMIT_ACK :
      remQueue(msg.act, msg.size, queue[pos], &queue_count[pos]);
      confirmed = 0;
      break;
    }
```

```
  } while (!confirmed);
}

void doQueue(u8_t seq, u8_t data)
{
  message_t msg;
  action_t act;
  int i;

  msg.type = QUEUE_REQ;
  msg.size = 1;
  msg.act = &act;
  act.addr.ip = 0;
  act.addr.port = 0;
  act.seq = seq;
  act.data = data;

  for (i = 0; i < target_count; i++) {
    send(&target[i], &msg);
  }

  for (i = 0; i < target_count; i++) {
    doConfirm(&act);
  }
}

int needCommit(u8_t seq, u8_t data)
{
  action_t act;
  int i, present;

  act.addr.ip = 0;
  act.addr.port = 0;
  act.seq = seq;
  act.data = data;

  for (i = 0; i < target_count; i++) {
    present = inQueue(&act, 1, queue[i], queue_count[i]);
    if (present) return 1;
  }

  return 0;
}

void sendCommit(int size)
{
  message_t msg;
  action_t act[CONC_CONN];
  int i;

  msg.type = COMMIT_REQ;
  msg.size = size;
  msg.act = act;

  copyQueue(queue[0], act, size);
  sortQueue(act, size);
```

```
  for (i = 0; i < target_count; i++) {
    send(&target[i], &msg);
  }
}

int tryCommit(void)
{
  int i, j, n, match;

  for (i = 0; i < CONC_CONN; i++) {
    for (j = 0; j < target_count; j++) {
      n = (j + 1) % target_count;
      match = inQueue(queue[j], queue_count[j], queue[n], queue_count[n]);
      if (!match) break;
    }
    if (match) {
      sendCommit(i);
      return 1;
    }
  }

  return 0;
}

void doCommit(u8_t seq, u8_t data)
{
  address_t src;
  message_t msg;
  action_t act[CONC_CONN];
  int i, confirmed, pos;
  int finished = 0, pending = 0;

  msg.act = act;

  do {
    printf("incoming...");
    recv(&src, &msg);
    printf("check!\n");

    pos = targetPosition(&src);
    if (pos < 0) error("POSITION");

    switch (msg.type) {
    case QUEUE_ACK :
      addQueue(msg.act, msg.size, queue[pos], &queue_count[pos]);
      finished = 0;
      if (!pending) pending = tryCommit();
      break;
    case COMMIT_ACK :
      remQueue(msg.act, msg.size, queue[pos], &queue_count[pos]);
      finished = !needCommit(seq, data);
      if (!finished && !pending) pending = tryCommit();
      break;
    }
  } while (!finished);

  printf("passed");
```

```
  for (i = 0; i < target_count; i++) {
    queue_count[i] = 0;
  }
}

int main(int argc, char *argv[])
{
  int i, err;
  message_t msg;
  action_t act[2];
  u8_t seq, data;

  if (argc <= 1) {
    printf("usage: %s <ip>:<port> ...\n", argv[0]);
    exit(-1);
  }

  if (argc > CONC_CONN + 1) {
    printf("error: too many target addresses specified\n");
    exit(-1);
  }

  for (i = 1; i < argc; i++) {
    err = readAddress(argv[i], &target[i - 1]);
    if (err) {
      printf("error: malformed target address\n");
      exit(-1);
    }
    target_count++;
  }

  queue = malloc(sizeof(action_t) * CONC_CONN * target_count);
  if (queue == NULL) error("MALLOC");

  queue_count = malloc(sizeof(int) * target_count);
  if (queue_count == NULL) error("MALLOC");

  for (i = 0; i < target_count; i++) {
    queue_count[i] = 0;
  }

  seq = 2;
  data = 222;

  doQueue(seq, data);
  doCommit(seq, data);

  return 0;
}
```

B.6 target.c

```
#include "default.h"
#include "channel.h"
#include "util.h"
```

```
action_t queue[CONC_CONN];
int queue_count = 0;

int value = 0;

void doQueue(address_t *src, message_t *msg)
{
  int i;

  msg->type = QUEUE_ACK;

  for (i = 0; i < queue_count; i++) {
    send(&queue[i].addr, msg);
  }

  addQueue(msg->act, msg->size, queue, &queue_count);

  msg->size = queue_count;
  msg->act = queue;

  send(src, msg);
}

void doCommit(address_t *src, message_t *msg)
{
  int queued, i;

  queued = inQueue(msg->act, msg->size, queue, queue_count);
  if (!queued) return;

  for (i = 0; i < msg->size; i++) {
    value = msg->act[i].data;
  }

  msg->type = COMMIT_ACK;
  for (i = 0; i < queue_count; i++) {
    send(&queue[i].addr, msg);
  }

  remQueue(msg->act, msg->size, queue, &queue_count);
}

int main(int argc, char *argv[])
{
  address_t src;
  message_t msg;
  action_t act[CONC_CONN];

  msg.act = act;

  while(1) {
    recv(&src, &msg);

    switch (msg.type) {
      case QUEUE_REQ :
        doQueue(&src, &msg);
        break;
```

```
      case COMMIT_REQ :
        doCommit(&src, &msg);
        break;
      default :
        error("TYPE");
    }
  }

  return 0;
}
```

B.7 util.h

```
void copyQueue(action_t *queue_a, action_t *queue_b, int size);
void addQueue(action_t *a, int size_a, action_t *b, int *size_b);
void remQueue(action_t *a, int size_a, action_t *b, int *size_b);
void sortQueue(action_t *queue, int size);

int inQueue(action_t *queue_a, int size_a, action_t *queue_b, int size_b);

void printAction(action_t *act);
void printQueue(action_t *act, int size);
void printMessage(message_t *msg);
```

B.8 util.c

```
#include "default.h"
#include "util.h"

int equalAddresses(address_t *a, address_t *b)
{
  return (a->ip == b->ip) && (a->port == b->port);
}

int equalActions(action_t *a, action_t *b)
{
  return equalAddresses(&a->addr, &b->addr) &&
    (a->seq == b->seq) && (a->data == b->data);
}

void copyAction(action_t *a, action_t *b)
{
  memcpy(b, a, sizeof(action_t));
}

void swapAction(action_t *a, action_t *b)
{
  action_t act;

  copyAction(a, &act);
  copyAction(b, a);
  copyAction(&act, b);
}

void copyQueue(action_t *queue_a, action_t *queue_b, int size)
{
  int i;
```

```
  for (i = 0; i < size; i++) {
    copyAction(&queue_a[i], &queue_b[i]);
  }
}

void addQueue(action_t *queue_a, int size_a, action_t *queue_b, int *size_b)
{
  int i;

  if (size_a + *size_b > CONC_CONN) error("ADD_QUEUE");

  for (i = 0; i < size_a; i++) {
    copyAction(&queue_a[i], &queue_b[*size_b]);
    *size_b = *size_b + 1;
  }
}

void remQueue(action_t *queue_a, int size_a, action_t *queue_b, int *size_b)
{
  int i, j, shift;

  for (i = 0; i < size_a; i++) {
    shift = 0;

    for (j = 0; j < *size_b; j++) {
      if (equalActions(&queue_a[i], &queue_b[j])) {
        shift++;
      } else {
        copyAction(&queue_b[j], &queue_b[j - shift]);
      }
    }

    *size_b = *size_b - shift;
  }
}

void sortQueue(action_t *queue, int size)
{
  action_t act;
  int i, j;

  for (i = 0; i < size; i++) {
    for (j = i; j < size; j++) {
      if (queue[i].addr.ip < queue[j].addr.ip) {
        break;
      }
      if (queue[i].addr.ip > queue[j].addr.ip) {
        swapAction(&queue[i], &queue[j]);
        break;
      }
      if (queue[i].addr.port > queue[j].addr.port) {
        swapAction(&queue[i], &queue[j]);
      }
    }
  }
}
```

```
int inQueue(action_t *queue_a, int size_a, action_t *queue_b, int size_b)
{
  int i, j, found;

  for (i = 0; i < size_a; i++) {
    found = 0;

    for (j = 0; j < size_b; j++) {
      if (equalActions(&queue_a[i], &queue_b[j])) {
        found = 1;
        break;
      }
    }

    if (!found) return 0;
  }

  return 1;
}

void printAction(action_t *act)
{
  printf("act_ip: %s\n", inet_ntoa(act->addr.ip));
  printf("act_port: %u\n", ntohs(act->addr.port));
  printf("act_seq: %u\n", act->seq);
  printf("act_data: %u\n", act->data);
}

void printQueue(action_t *act, int size)
{
  int i;

  for (i = 0; i < size; i++) {
    printf("act_pos: %d\n", i);
    printAction(&act[i]);
  }
}

void printMessage(message_t *msg)
{
  int i;

  printf("act_type: %u\n", msg->type);
  printf("act_size: %u\n", msg->size);

  for (i = 0; i < msg->size; i++) {
    printf("act_pos: %d\n", i);
    printAction(&msg->act[i]);
  }
}
```

C Promela code of the redundant algorithm

```
#define SERVERS 3
#define CLIENTS 2
#define CONCURRENT 1
```

```
#define CHAN_COUNT 25
#define CHAN_SIZE 1
#define MAX_OPERATIONS 2
#define MAX_STEPS 3
#define MAX_IDS 2

#define CHANNEL(from, to) \
    channels[(from - 1) + (to - 1) * (SERVERS + CLIENTS)]

c_code {
   #include "types.h"
   #include "minix.c"
}

int readers;
int writers;

typedef store_t {
   int value;
   int current;
   int concurrent;
}

chan channels[CHAN_COUNT] = [CHAN_SIZE] of {store_t};

inline clean(store) {
   store.value = 0;
   store.current = 0;
   store.concurrent = 0
}

inline send_msg() {
   CHANNEL(my_id, node_id)!out
}

inline receive_msg() {
   assert(CLIENTS == 2);

   end: if
   :: CHANNEL(SERVERS + 1, my_id)?in ->
      node_id = SERVERS + 1
   :: CHANNEL(SERVERS + 2, my_id)?in ->
      node_id = SERVERS + 2
   fi;

   c_code { merge(&Pserver->in, &Pserver->out); };
   clean(in);
}

inline scatter_msg() {
   assert(MAX_IDS <= SERVERS);
   assert(MAX_IDS >= SERVERS / 2 + 1);

   assert(count_ids == 0);
   assert(skip_ids == 0);

   do
```

```
    :: count_ids - skip_ids < MAX_IDS ->
       CHANNEL(my_id, count_ids + 1)!out;
       count_ids++
    :: skip_ids + MAX_IDS < SERVERS ->
       count_ids++;
       skip_ids++;
    :: else -> break
    od;

    count_ids = 0;
    skip_ids = 0
}

inline gather_msg() {
    assert(SERVERS == 3);

    assert(count_ids == 0);

    do
    :: count_ids < MAX_IDS ->
       if
       :: CHANNEL(1, my_id)?in
       :: CHANNEL(2, my_id)?in
       :: CHANNEL(3, my_id)?in
       fi;
       count_ids++;

       c_code { merge(&Pclient->in, &Pclient->store); };
       clean(in);

    :: else -> count_ids = 0; break
    od
}

inline process() {
    receive_msg();

    c_code { merge(&Pserver->in, &Pserver->out); };

    send_msg()
}

inline read() {
    assert(count_steps == 0);

    do
    :: count_steps < MAX_STEPS ->
       scatter_msg();
       gather_msg();
       count_steps++;

       if
       :: c_expr { read_done(&Pclient->store, &Pclient->out) } -> break
       :: else -> c_code { read_step(&Pclient->store, &Pclient->out); }
       fi

    :: else -> break
```

```
    od;

    count_steps = 0
}

inline write() {
    assert(count_steps == 0);

    do
    :: count_steps < MAX_STEPS ->
        scatter_msg();
        gather_msg();
        count_steps++;

        if
        :: c_expr { write_done(&Pclient->store, &Pclient->out) } -> break
        :: else -> c_code { write_step(&Pclient->store, &Pclient->out); }
        fi

    :: else -> break
    od;

    count_steps = 0
}

proctype server(int my_id) {
    store_t in, out;
    int node_id;

    restart: do
    :: process()
    od
}

proctype client(int my_id) {
    store_t in, store, out;
    int count_ids, skip_ids;
    int count_ops;
    int count_steps;

    restart: do
    :: count_ops < MAX_OPERATIONS ->
        if
        :: atomic { CONCURRENT || writers == 0 ->
            readers++ };
            read();
            readers--;
        :: atomic { CONCURRENT || readers + writers == 0 ->
            writers++ };
            if
            :: out.value = 0; write()
            :: out.value = 1; write()
            fi;
            writers--
        fi;
        count_ops++
    :: else -> break
```

```
   od;
}

init {
   int n;

   atomic {
      do
      :: n < SERVERS ->
         run server(n + 1);
         n++
      :: else -> break
      od;

      do
      :: n < CLIENTS + SERVERS ->
         run client(n + 1);
         n++
      :: else -> break
      od
   }
}
```

D MINIX code of the redundant algorithm

D.1 channel.h

```
#include "types.h"

#ifndef CHANNEL
#define CHANNEL

int open_ether(void);
void read_ether(int fd, ether_addr_t *src, store_t *store);
void write_ether(int fd, ether_addr_t *dst, store_t *store);

#endif
```

D.2 channel.c

```
#include "default.h"
#include "types.h"

#define DEFAULT_DEVICE "/dev/eth"
#define ETH_TYPE 0x2222

int open_ether(void)
{
   int fd, err;
   char *eth_device;

   nwio_ethopt_t nwio_ethopt;

   if ((eth_device = getenv("ETH_DEVICE")) == NULL)
      eth_device = DEFAULT_DEVICE;
   fd = open(eth_device, O_RDWR);
```

```
    if (fd < 0) error("ETH_DEVICE");

    nwio_ethopt.nweo_type = ETH_TYPE;

    nwio_ethopt.nweo_flags = NWEO_EXCL | NWEO_EN_LOC | NWEO_DI_BROAD;
    nwio_ethopt.nweo_flags |= NWEO_DI_MULTI | NWEO_DI_PROMISC | NWEO_REMANY;
    nwio_ethopt.nweo_flags |= NWEO_TYPESPEC | NWEO_RWDATALL;

    err = ioctl(fd, NWIOSETHOPT, &nwio_ethopt);
    if (err) error("NWIOSETHOPT");

    return fd;
}

void read_ether(int fd, ether_addr_t *src, store_t *store)
{
    int count;
    char buffer[ETH_MAX_PACK_SIZE];
    message_t *message;

    count = read(fd, &buffer, ETH_MAX_PACK_SIZE);
    if (count != ETH_MIN_PACK_SIZE) error("READ_ETHER");

    message = (message_t*)buffer;

    memcpy(src, &message->eth_hdr.eh_src, sizeof(ether_addr_t));
    memcpy(store, &message->store, sizeof(store_t));
}

void write_ether(int fd, ether_addr_t *dst, store_t *store)
{
    int err, count;
    char buffer[ETH_MIN_PACK_SIZE];
    message_t *message;
    nwio_ethstat_t nwio_ethstat;

    err = ioctl(fd, NWIOGETHSTAT, &nwio_ethstat);
    if (err) error("NWIOGETHSTAT");

    message = (message_t*)buffer;

    message->eth_hdr.eh_src = nwio_ethstat.nwes_addr;
    message->eth_hdr.eh_dst = *dst;
    message->eth_hdr.eh_proto = ETH_TYPE;

    message->store = *store;

    count = write(fd, message, ETH_MIN_PACK_SIZE);
    if (count != ETH_MIN_PACK_SIZE) error("WRITE_ETHER");
}
```

D.3 default.h

```
#define _MINIX_SOURCE 1

#include <unistd.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <stdarg.h>
#include <stddef.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/select.h>
#include <net/hton.h>
#include <net/gen/ether.h>
#include <net/gen/eth_io.h>
#include <net/gen/eth_hdr.h>
#include <net/gen/if_ether.h>

#define CONC_CONN 32

#include "types.h"

void error(char *msg);
void print_store(store_t *store);
```

D.4 default.c

```
#include "default.h"

void error(char *msg)
{
    printf("%s: %s\n", msg, strerror(errno));
    exit(-1);
}

void print_store(store_t *store)
{
    printf("store: val=%d, cur=%d, conc=%d\n", store->value, store->current, store->concurre
}
```

D.5 client.c

```
#include "default.h"
#include "channel.h"
#include "types.h"
#include "minix.h"

#define MAX_STEPS 4

store_t store, out;

void scatter_ether(int fd, ether_addr_t *dst, int dst_size, store_t *out)
{
    int i;

    for (i = 0; i < dst_size; i++) {
        write_ether(fd, &dst[i], out);
    }
}
```

```
void gather_ether(int fd, int dst_size, store_t *store)
{
   int i;
   ether_addr_t src;
   store_t in;

   for (i = 0; i < dst_size; i++) {
      read_ether(fd, &src, &in);
      merge(&in, store);
   }
}

int read_value(int fd, ether_addr_t *dst, int dst_size)
{
   int i;

   for (i = 0; i < MAX_STEPS; i++) {
      scatter_ether(fd, dst, dst_size, &out);
      gather_ether(fd, dst_size, &store);

      if (read_done(&store, &out)) return store.value;
      read_step(&store, &out);
   }

   return -1;
}

int write_value(int fd, ether_addr_t *dst, int dst_size)
{
   int i;

   for (i = 0; i < MAX_STEPS; i++) {
      scatter_ether(fd, dst, dst_size, &out);
      gather_ether(fd, dst_size, &store);

      if (write_done(&store, &out)) return 0;
      write_step(&store, &out);
   }

   return -1;
}

int main(int argc, char *argv[])
{
   int i, fd, err, retval;
   ether_addr_t src, dst[CONC_CONN], *addr;

   if (argc < 3) error("ARGC");

   for (i = 2; i < argc; i++) {
      addr = ether_aton(argv[i]);
      if (addr == NULL) error("ETHER_ATON");
      dst[i - 2] = *addr;
   }

   fd = open_ether();
```

```
    store.value = 0;
    store.current = 0;
    store.concurrent = 0;

    out.value = 0;
    out.current = 0;
    out.concurrent = 0;

    if (argv[1][0] == 'r') {
        retval = read_value(fd, dst, argc - 2);
    } else {
        out.value = atoi(argv[1]);
        retval = write_value(fd, dst, argc - 2);
    }

    close(fd);

    return retval;
}
```

D.6 minix.h

```
#ifndef MINIX
#define MINIX

void merge(store_t *in, store_t *out);

void read_step(store_t *store, store_t *out);
int read_done(store_t *store, store_t *out);

void write_step(store_t *store, store_t *out);
int write_done(store_t *store, store_t *out);

#endif
```

D.7 minix.c

```
#include "default.h"
#include "types.h"
#include "minix.h"

#define MAX(a, b) (a > b ? a : b)
#define TRUE 1
#define FALSE 0

void merge(store_t *in, store_t *out) {
    if (in->value != out->value) {
        if (in->current > out->current) {
            out->concurrent = MAX(in->concurrent, out->current);
        } else {
            out->concurrent = MAX(in->current, out->concurrent);
        }
    } else {
        out->concurrent = MAX(in->concurrent, out->concurrent);
    }

    if (in->current > out->current) {
```

```
        out->value = in->value;
        out->current = in->current;
    }
}

void read_step(store_t *store, store_t *out) {
    out->value = store->value;
    out->concurrent = store->concurrent;

    if (store->current == out->current) {
        out->current = store->current + 1;
    } else {
        out->current = store->current;
    }
}

int read_done(store_t *store, store_t *out) {
    if (store->current == store->concurrent + 2) {
        return store->current == out->current;
    }

    return store->current > store->concurrent + 2;
}

void write_step(store_t *store, store_t *out) {
    out->concurrent = store->concurrent;

    if (store->current == out->current) {
        out->current = store->current + 1;
    } else {
        out->current = store->current;
    }
}

int write_done(store_t *store, store_t *out) {
    if (store->value != out->value) {
        return FALSE;
    } else {
        return read_done(store, out);
    }
}
```

D.8 server.c

```
#include "default.h"
#include "channel.h"
#include "types.h"
#include "minix.h"

int main(int argc, char *argv[])
{
    int fd;
    ether_addr_t src;
    store_t in, out;

    out.value = 0;
    out.current = 0;
```

```
    out.concurrent = 0;

    fd = open_ether();

    for(;;) {
        read_ether(fd, &src, &in);
        merge(&in, &out);
        write_ether(fd, &src, &out);
    }

    close(fd);

    return 0;
}
```

D.9 types.h

```
#ifndef TYPES
#define TYPES

typedef struct {
    int value;
    int current;
    int concurrent;
} store_t;

typedef struct {
    eth_hdr_t eth_hdr;
    store_t store;
} message_t;

#endif
```