

On Distributed Algorithms in a Broadcast Domain ^{*}

Danny Dolev^{**}, Dalia Malki^{***}

The Hebrew University of Jerusalem, Israel

Abstract. This paper studies the usage of broadcast communication in distributed services. The approach taken is practical: all the algorithms are asynchronous, and tolerate realistic faults. We study four problems in a broadcast domain: clock synchronization, reliable and ordered broadcast, membership, and file replication. The clock synchronization algorithm shows for the first time how to utilize broadcast communication for synchronization. The master synchronizes any number of slaves while incurring a constant load. The approach taken in the file replication tool uses *snooping* in order to enhance the availability of file systems, at almost no cost.

1 Introduction

This paper presents algorithms that use broadcast communication. The broadcast primitive enables the dissemination of messages to multiple destinations via a single transmission. The motivation behind this work is practical: most computer networks nowadays essentially provide a datagram broadcast service. Most transport protocols do not utilize the broadcast capability, though, because the handling of faults and retransmissions is far more complicated than point to point communication. Future networks designs also appear to possess the broadcast capability (*e.g.* FDDI, MAN, wireless networks based on cellular communication).

Thus, it is important to define high level services over the datagram broadcast layer, and examine how applications can utilize broadcast communication. Our work incorporates the broadcast primitive into the system model. We show various algorithms in this model that are substantially different from their point-to-point counterparts. We focus on practical algorithms, and present their basic properties. Therefore, all the algorithms are asynchronous, and tolerate realistic faults.

The first algorithm we present is for a clock synchronization service. While it is rather obvious that broadcast messages are capable of carrying *data* to multiple destinations efficiently, we have not seen any use of it in synchronization. We

^{*} This paper appears in Intl. Conference on Automata, Languages and Programming, July 1993, Lund Sweden, pp. 371-387.

^{**} also at IBM Almaden Research Center

^{***} This work was supported in part by GIF I-207-199.6/91.

show how multiple clients can synchronize with a single time-source (“master”) using broadcast messages. The master incurs a constant overhead of emitting the broadcast messages, regardless of the number of clients being synchronized.

The second service presented is a reliable broadcast service, that delivers messages to a group of processes, while preserving their relative orders. This paper surveys some of the protocols suggested in the literature for these services. One of the interesting trade-offs manifested in these protocols is whether to employ a central coordinator for achieving an agreed order of delivery of the messages.

Another category of distributed algorithms that are examined in the broadcast domain are algorithms that need to achieve coordinated decisions. The impossibility result of [14] bounds all these problems within an asynchronous environment. The problem is the inability to distinguish between a slow machine and one that crashed. In practical asynchronous systems it is often preferable to give-up on a slow machine, rather than get stuck in waiting.

The membership protocol we present circumvents this problem by maintaining the set of machines that appear active *internally* at any point. Moreover, other protocols can use the membership service instead of explicitly handling the dynamicity in the system. The specific membership protocol presented here handles both failures and recoveries of machines, and network partitions and merges. We provide an informal definition of the requirements of the membership service, while allowing partitions.

We conclude with a utility that exploits an unusual facet of broadcast communication, the ability to *snoop* and intercept broadcast messages. We show how to use the snooping ability in order to enhance availability of file system, at almost no cost.

None of the above protocols explicitly uses any known theoretical solution to coordination problems, even though these problems were extensively studied. Randomized protocols prove their usefulness in overcoming the impossibility of consensus ([14]). However, they proved to be too complicated for usage, and typically exchange too many messages. Specifically to this context, we note that there are no randomized protocols that utilize available broadcast as a primitive.

Another approach that was taken in many works, is to assume that the system is synchronized. Thus, all machines perform their operations in a synchronous lock step. There were a few variations to this approach, but shared by all of them is the need to guarantee a tight synchronization. In practice this is not a valid approach, since distance, temporary load, and the independence in operation of individual machines prevents us from guaranteeing such a synchronization. It is true that one can assume, for instance, that 90% of the messages arrive within a small window of time, but the rest of the messages may take a much longer time to arrive. Tight synchronization requires a single step of the protocol to be very large, so that all but few of the messages sent at that step will arrive correctly.

The algorithms presented in this paper operate within a realistic model, and were all implemented. We encourage other researchers to continue investigating the possibilities and tradeoffs within this framework, and explore the possibility

of using randomized techniques in practice.

2 System Model

This section presents the assumed underlying model of the rest of the paper. We believe that the model below reflects the basic concepts shared by most distributed systems today. A distributed system comprises of a set of machines, that communicate using messages. The underlying network is completely connected and can transfer messages between any pair of machines. In addition, the network provides a broadcast service. A machine emits a broadcast message at once to all its destinations. A single copy of the message reaches all the machines, but might arrive at different times to them.

An essential property of the model is that broadcast messages incur the same processing overhead as unicast messages at the sending and at the receiving machines. Similarly, broadcast messages consume identical network capacity as unicast messages.

A broadcast message is sent “anonymously” and does not require specific addressees. In this paper we will not distinguish between broadcast, which sends a message to all the machines in the network, and a *multicast*, which sends a message to a selected subset via a group-designation. The broadcast capability also enables in some cases a non-targeted machine to *snoop* and intercept messages. This might be a security concern for some environments; in this paper we present an advantageous facet of this property.

A typical LAN architecture is composed of one or more broadcast segments, interconnected via bridging and gateway elements. The LAN might partition to two sets of machines, in case a bridging/gateway element fails. In case that some other machine fails or disconnects from the network, the remaining machines continue to be connected and to pass messages undisturbed. Machine failure is either fail stop or omission, thus a machine may fail to send messages, but will not produce messages that are not part of the basic protocol. Thus, no byzantine faults are assumed. Moreover, in the context of this paper it is assumed that if a message arrives to its destination, its data is assumed to be uncorrupted.

The broadcast service provided by the hardware media is an unreliable datagram service. A single transmission can potentially reach all the machines, but it may fail to go out, or a single machine might miss it. Although message loss rate is not specified explicitly, the protocols were designed under the assumption that it is low.

The Basic Impossibility Result

At the basis of any coordination problem in a distributed system lies some algorithm like atomic-broadcast and membership-maintenance which requires a consensus at one level or another. Fischer Lynch and Paterson ([14]) were the first to point out that there is no way to reach consensus in an asynchronous distributed system, when faults may occur. Moreover the asynchrony that produces the difficulty can be very limited, as can be seen in [11].

The basic idea behind all these impossibility results is that there is no way to distinguish between a very slow machine and a failed one. Since any nontrivial coordination problems should be determined on the fly according to the initial inputs to the individual machines, there should always be a case in which a single machine may determine the general outcome. Thus, as long as the input from this machine does not arrive, we cannot guarantee that all the correctly operating machines will perform the same operation (or reach the same decision).

The Practical Approach

In practice, coordination problems cannot be solved in “full”, *i.e.* their solution may not comply with an outsider’s view of the system and its view of “correct” processors. In order to be able to overcome the impossibility result, machines will use inaccurate fault detectors, based on timeout. The coordination decisions are reached only among the machines that are viewed *internally* as operational, and not necessarily “really” all the operational machines. Thus, in the cases that the fault detector errs, a correctly operating machine may be considered faulty, and will need to reconcile with the rest of the machines at some later time.

The basic differences can be summarized as follows: in our approach individual machines operate asynchronously, each one at its own pace. Prior to some decision points some machine may need to wait for others, but instead of a deadlock, a timeout will detect failures, and will enable the machines to resume operation.

3 Clock Synchronization

In this section we will describe a new approach to clock synchronization. The approach utilizes the broadcast domain environment and enables synchronizing many machines without explicitly exchanging messages with each one of them.

Common to all clock synchronization algorithms is the need to exchange messages between the synchronizer and the synchronized machine. This is true in master-slave protocols, as well as in all versions of distributed clock synchronization protocols. The need arises from the requirement to produce three events, two on one clock and a middle one on a different clock. Once such events are identified, one can obtain an estimate on the difference between the clock readings. This estimate is used to adjust one of the two clocks.

An access to an outside time source is a common way to obtain a precise time service. And several researchers have offered to use the machine having that input as the master clock that will occasionally synchronize the rest.

This approach seems to mean that in such a time service, if many machines are connected via a LAN, and one of them is the master, it will need to explicitly exchange messages with each one of them in order to synchronize all the clocks. When the LAN contains dozens of machines this becomes a real load on the master machine. The problem is to find a way to broadcast the time information to synchronize individual clocks without the need to explicitly exchange messages between the master and the slaves.

A Simplified Solution

First concentrate on a simplified model. Assume that the master machine has a direct access to the LAN in the sense that it does not use any buffer to store messages it sends out. Moreover, assume that there are no other messages produced at the master machine. Thus all it produces are clock synchronization messages.

Thus, when it decides to send a message it knows when this message is being sent out. Thus it can produce a sequence of messages, each one being produced after the previous one was actually sent out. The master will timestamp each message in such a sequence, and will broadcast it over the LAN.

Ignoring message loss, for a moment, we can study the flow of messages at each receiving machine. These machines naturally are busy with other functions, and when a machine reads its input, it may find several messages of the sequence at its input buffer at once. A machine needs to identify two events of the master that took place between one of its own events.

The receiving machine marks the time it first notices a partial sequence sent by the master at its buffer. Let this time be R_1 and let x_1, \dots, x_k be the prefix of the sequence it noticed at its input buffer. At a later time, say R_2 it reads its input again and sees some more messages of the sequence sent by the master. Let this new subsequence be denoted x_{k+1}, x_{k+2}, \dots . The simple model assures that each one of these messages was produced after the previous one was sent. On the other hand, we do not know that x_{k+1} was produced after time R_1 , since it is possible that this message was in transit when the receiver was reading its input buffer. But in this case we know that x_{k+2} was not produced yet, thus its timestamp is an event that took place after time R_1 and before time R_2 . This implies that the timestamp on this message, say T_{k+2} , can be used to synchronize the receiver clock. Figure 1 exemplifies these events.

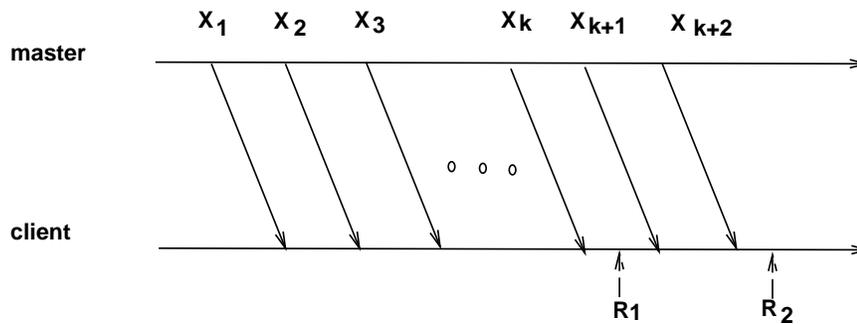


Fig. 1. Event x_{k+2} occurs between R_1 and R_2

Adapting ideas from previous clock synchronization protocols, the receiver

can adjust its clock by,

$$R_2 - T_{k+2} + \frac{(R_2 - R_1)}{2}.$$

Notice that all receivers on the LAN can synchronize using the same method. Thus, the load on the master does not depend on the number of machines that are currently connected.

The general case

In the general case there may be specific architectures in which more than one message might be in transit at once. In such architectures, the number of such messages is still bounded by a small constant number. The basic idea above can easily be adopted to count for that.

There might be cases where the master doesn't necessarily know when its messages are being sent out. In this case we can borrow another idea from existing clock synchronization protocols. We can assign a specific machine to echo back to the master whenever it receives each message. The master waits for an echo on its previous message before time-stamping and sending the next one. In a sense, the master and this machine follow the basic idea of a master-slave protocol. The rest of the receivers synchronize their clocks just by listening in.

In the [13] the reader can find the complete study of this approach and various optimizations to the problem.

4 Reliable Broadcast

The topic of consistent dissemination of information in distributed systems has been the focus of many studies, both theoretical and practical. The pioneering work of the V system ([8]), deals with communication among groups of processes, via broadcast messages. In V, broadcast messages are not reliable, but provide "best effort" delivery semantics. In addition, if messages are sent concurrently from several sources, the order of their delivery at overlapping destinations is undefined. Later work in the ISIS system ([6]), deals with providing higher level services, and supports reliable delivery, as well as various orderings. Many distributed applications require such high degree of coordination among their processes. The main difficulty facing the designer of a distributed application is the consistency of information disseminated, and the control over the dissemination of that information. Thus, the designer of a distributed system would wish for a service that provides a guaranteed delivery-and-consistency of broadcast messages. Having such a service, most distributed applications become much easier to implement and to maintain.

In many systems, when a group of processes need to perform a coordinated work they interact via (reliable) point-to-point communication. This approach is costly when there are several participants. It would be preferable to use the

available broadcast hardware where possible, for efficient dissemination of messages to multiple destinations via a single transmission. The problem is that current transport protocols provide only datagram broadcast services (*e.g.* UDP [22], IP-multicast [10]).

Today, there are several projects that develop protocols for reliable broadcast services while utilizing the broadcast hardware where possible, *e.g.* [18, 17, 3, 2] and the recent version of the ISIS system [7]. We discuss some of them in this section. In this section it is assumed that the system consists of a static set of machines (the Membership Section shows how to maintain the set of *active* machines up to date, and the protocols we present below can be extended to dynamic environments once the membership layer is present).

Causal Broadcast

This section presents the mechanism employed in the Transis communication sub-system ([2]) for guaranteeing delivery of messages to all their destinations. The principle idea of reliable message delivery in Transis is motivated by the Trans algorithm ([18]) and the Psync algorithm ([21]).

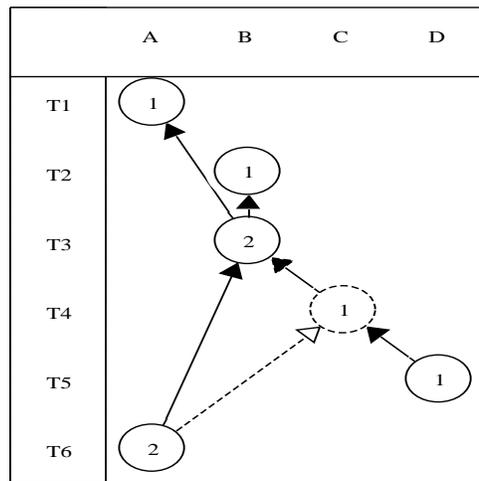


Fig. 2. A Transis Scenario

Messages are transmitted via a single transmission, using the available network broadcast. The “blobs” in Figure 2 represent broadcast messages. Each machine tags its messages with increasing serial numbers, serving as message-ids. For example, in the figure, machine A emits at the time-mark T1 the first

message, machine B emits at T2 its first message, and so on. Acknowledgments to messages are piggybacked onto the next broadcast messages. The full arrows represent acknowledgments: from message B_2 to A_1 and to B_1 , from C_1 to B_2 , etc. An ACK consists of the sending machine-id and the serial number of the acknowledged message. A fundamental principle of the protocol is that each ACK need only be sent once. Further messages, that follow from other machines, form a “chain” of ACKs, which implicitly acknowledge former messages in the chain. For example, Figure 2 could depict the following scenario on the network:

$$A_1, B_1, \xrightarrow{A_1 B_1} B_2, \xrightarrow{B_2} C_1, \xrightarrow{C_1} D_1, \dots$$

Machines on the LAN might experience message losses. They can recognize it by analyzing the received message chains. For example, machine A recognized that it lost C_1 after receiving the sequence: $A_1, B_1, \xrightarrow{A_1 B_1} B_2, \xrightarrow{C_1} D_1$. Therefore, A emits a *negative-ACK* on message C_1 , requesting for its retransmission. In this case A acknowledges B_2 and not D_1 , since messages that follow “causal holes” are not incorporated for delivery until the lost messages are recovered. In this way, the acknowledgments form the causal relation among messages directly.

The delivered messages are held for backup by all the receiving machines. In this way, retransmission requests can be honored by any one of the participants. Of course, messages cannot be kept for retransmission forever. When all the machines have acknowledged the reception of a message, it can be safely discarded.

If the LAN runs without losses then it determines a single total order of the messages. Since there are message losses, and machines receive retransmitted messages, the original total order is lost. The piggybacked acknowledgments are used for reconstructing the original partial order of the messages.

Agreed Broadcast

One of the characteristics of the Trans and the Transis protocols, is that they allow completely spontaneous transmission of messages by any machine. Consequently, two machines may send messages within a small interval apart, none receiving each other’s message first. In this case, there will be no acknowledgment between these messages. This means that additional processing is required if there is a requirement to deliver the messages in the same total order at all their common destinations.

An *agreed broadcast* service guarantees that messages arrive reliably and in the same total-order to all their destinations. There are several completely distributed algorithms that build a total order from the local information and reach agreement ([18, 12, 21]). It is perhaps easiest to understand a naive *all-ack* algorithm that is also completely distributed. The above referred algorithms are essentially optimizations on this principle. The all-ack idea is:

- Wait until at least one message is received from each machine.

- Then go through the machines in ascending order, and deliver the first message from each machine unless it directly acknowledges another message.

The common characteristic of these algorithms, is that they do not incur any extra message exchange for achieving agreement on the total order. They have *post-transmission* delay, from the time a message is transmitted and received until it is ordered in the right place. Interestingly, this cost is most apparent when the system is relatively idle, and waiting for responses from all (or some) of the machines incurs the worst-case delay. On the other hand, these methods can sustain steady transmission loads that are close to the network limits, when all the machines are fairly uniformly active (*e.g.* the ToTo protocol was measured delivering around 500 1K messages per second over an Ethernet of 10 Indigo stations, see [12]).

A different family of protocols orders the messages in a total order by employing a centrally controlled ordering scheme ([7, 3, 17]). The Isis ABCAST protocol ([7]) employs a *token-holder* within each group of communicating processes. ABCAST messages are broadcast at will, and their delivery is delayed by all the receiving processes except for the token holder. Periodically, the token holder sends a message indicating its order of delivery for all received ABCAST messages, and all the other processes comply with it.

The Amoeba system contains a different variation of this scheme, implemented within the operating system kernel ([17]). A sequencer kernel is designated as the central controller. Every message is sent to it via point to point communication, and the sequencer broadcasts it to all the machines. The FIFO order of sequencer-transmissions determines a total order for all the messages.

The Totem protocol ([3]) uses a revolving token that holds a sequence-number for messages. The holder of the token can emit one or more broadcast messages, and update the token sequence accordingly. In order to transmit a broadcast message, a processor must obtain the token. The token itself regularly revolves among all the processors.

The cost in these protocols is in obtaining access to the central controller, be it a processor or a token. This cost is apparent both in the delay occurring until the control is obtained, and in extra messages exchanged. Once it is obtained, transmission and ordering is done immediately. Therefore, we say that they have a *pre-transmission* delay. The advantage of central control is that it regulates the flow of messages efficiently. It is not entirely clear what are the trade-offs between distributed and centralized control in these protocols. In particular, the behavior of these protocols when the communication pattern is “chaotic” need to be further investigated.

5 Membership in Broadcast Environments

A point to point communication protocol needs to maintain information about one machine, “the other party.” A reliable broadcast communication system needs to maintain information about a set of machines of a variable size. The

machines may fail and recover. The underlying communication network may partition and reconnect, thereby partitioning the set of participating machines. This dynamicity is one of the main reasons that reliable broadcast protocols are more complex than their point-to-point counterparts.

The *membership* problem is to maintain the set of participating machines in agreement among all the machines. This basic problem of distributed computing has received considerable attention in the past (see [9, 1, 19, 20, 23, 24, 16, 3, 4]). We are mainly interested in membership protocols for broadcast communication environments. In these environments, the membership changes are reported via special messages, that are delivered to the upper level application among the stream of regular messages.

In distributed applications, the machines typically act upon regular messages according to their installed membership. Thus, in addition to the agreement on membership changes, it is desired that the machines see the membership changes in the *same order*. Furthermore, in order for all the machines to respond in the same manner to broadcast messages, they should see the same messages between every pair of membership changes. This valuable principle is defined in [5], and is called *virtual synchrony*.

Informally, we require that membership changes maintain:

- Membership changes occur in the same order at all the machines that view them.
- Every failed or disconnected machine is removed from its membership within a finite time.
- Every two operational machines that are connected for sufficiently long time *join* in a common membership.
- Membership changes preserve virtual synchrony with respect to regular messages.

We briefly present a protocol that satisfies all these requirements here. The protocol relies on broadcast communication that preserves *causality*.⁴ This protocol is completely symmetrical. Joining with other machine(s) is triggered when a message from a machine that does not belong to the current membership view is intercepted in the broadcast domain. Fault handling is triggered by timeout. (A closely related membership protocol that satisfies the above requisites is presented in [1]).

Whenever the membership protocol starts, each machine sends a message with the best *suggestion* it has for the current membership. Each membership suggestion contains two sets: all the known machines, called M , and all the suspected faulty/detached machines, called F . In order to *accept* the membership suggested in $\langle M, F \rangle$, all the machines in $M \setminus F$ need to broadcast identical suggestions. If a membership suggestion $\langle M', F' \rangle$ from $M \setminus F$ differs from $\langle M, F \rangle$, then there are a few cases:

⁴ We say that two messages m, m' are related in the causal order \xrightarrow{cause} , if they are in the transitive closure of: (1) $m \xrightarrow{cause} m'$ if $deliver_q(m, *) \rightarrow broadcast_q(m')$, (2) $m \xrightarrow{cause} m'$ if $broadcast_q(m) \rightarrow broadcast_q(m')$

- If $M' \subseteq M$ and $F \subseteq F'$, then this message is ignored.
- If M' or F' contains machines that are not contained in M, F , and the sender of this suggestion did not agree already to $\langle M, F \rangle$, then M', F' are merged into M, F and a new membership suggestion is broadcast.
- If M' or F' contains machines that are not contained in M, F , and the sender of this message is already marked as agreeing to $\langle M, F \rangle$, then the message is queued for future membership instances. This handling is crucial for the consistency of the membership decision.

If there are machines in F' that are not included in F , they will not be required to agree to the $\langle M, F \rangle$ suggestion (this could lead to a deadlock). In this case, all the machines in $M \setminus (F \cup F')$ must send their agreement both to the $\langle M, F \rangle$ suggestion, and to the suspected machines in F' .

As shown in these cases, the suggestion of each participating machine may change during the execution of the membership protocol, one or more times. Therefore, this protocol cannot be classified as a k -phase protocol for any specific k , and the number of rounds of message exchanges depends on the specific scenario.

During an instance of the membership protocol, the suspected machines are **not** removed from M , but are only added to F . A machine that is suspected in F , cannot be removed from F either. This guarantees that the protocol will terminate within a finite time. For example, during a period of instability in the network, a certain machine might detach and re-connect frequently. The removal of this machine from M might lead to an endless process of removing and adding it to M . In our scheme, it can be added and removed at most once during the execution of the protocol. Consequently, our scheme might mistakenly remove from the membership an operational machine. This machine can later re-join the membership. Note that in an asynchronous environment, there is no way to prevent the removal of a slow machine from the membership. Thus, in our view, the means for reducing the potential of such mistakes are practical means: fine-tuning of the system timeouts, and a robust fault-detection mechanism, involving *consulting* with a few machines. These practical details are not relevant for the correctness of the membership protocol.

This protocol also preserves virtual synchrony with respect to other regular messages in the system. In order to understand the main difficulty in preserving virtual synchrony, envision a system of four machines, A, B, C, D. Machine D has crashed, and its last message m_d is received only by C. If C sends its membership suggestion $\langle \{A, B, C, D\}, \{D\} \rangle$ (for excluding D) before it receives m_d , how will A and B know they must deliver this message before the membership-change? There may be more complicated scenarios, for example if first D crashes, and C is the only receiver of m_d , and then C crashes, but has sent a message m_c referring to m_d . The rule for message delivery in our protocol is the following: Between every two membership changes, all the messages that follow *any one of the identical membership suggestion-messages of the first membership-change* and do not follow *any one of the identical membership suggestion-messages of the second membership-change* are delivered. This set of messages can be proved

to be identical among all the machines that install the same two membership-changes.

6 Warm Replication by Snooping

This section deals with a less obvious facet of broadcast communication, the ability to intercept messages by non-target machines (*snooping*). We propose a way to exploit this ability in order to enhance availability of system services. The snooping ability offers a novel way for cheaply replicating services in the network.

To exemplify our ideas, we use the Sun Network File System (NFS) environment, available at Unix environments. In an NFS environment, applications access files throughout the network in an automatic, transparent way. We can view the entire network as providing a global file system service that is distributed among different machines. While very convenient in all ways, this distribution leads to a reliability problem: The failure of any one of the machines that provide file-system services can block an application from running.

In these environments, local-area broadcast networks such as Ethernet and token-rings are becoming a standard de-facto. These broadcast media carry the point-to-point NFS messages and enable snooping by unlisted parties.

A *warm-backup service* (WB) provides a *per-application* replication service. The main mechanism of WB is quite simple: When an application asks for WB service, a second replica will be created for each file that the application opens. The *warm-backup service* will keep the two replicas up-to-date and consistent by snooping, and intercepting the file-modification messages. WB performs these changes on the replica. When that file is not needed any more, the new replica will be deleted. In order to enhance availability even more, the same scheme can work with any number of additional replicas for each file, instead of only one.

Another option of the WB service is to provide a *per-directory* warm replication service. When this option is specified for the WB server, then only files in the specified directory sub-hierarchy are automatically replicated. We anticipate this to be a most useful option for the WB tool.

The WB server differs from other known replication systems in that it provides a *per-application/per-directory* replication service in order to increase the accessibility of files throughout the application's lifetime. The main novelty is the use of *snooping* in broadcast environments for providing replication cheaply. It does not require any special hardware such as multiple-access disks, yet it provides warm replication that is consistent at every moment. In addition, unlike fully-replicated file systems, the WB architecture does not require modification to the basic file-system structure or semantics.

Concurrent-Write WB

In the general case, multiple processes from different machines may access the same file concurrently. In order to keep the primary file copy and its replica(s)

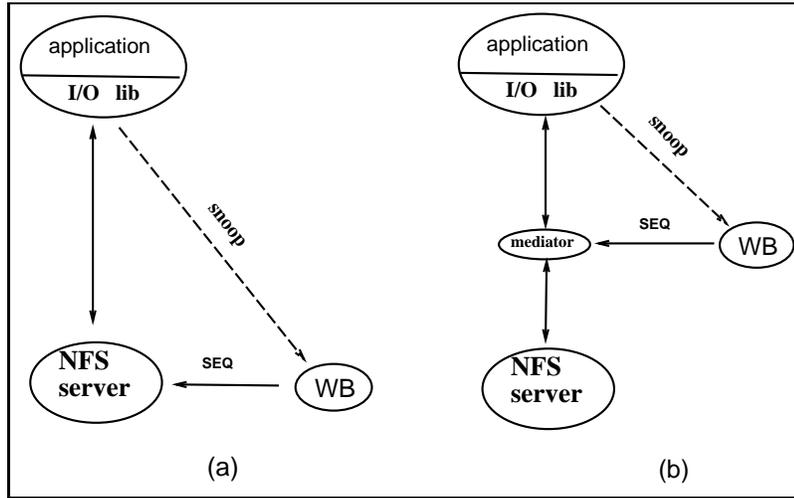


Fig. 3. Concurrent-Reader WB

consistent, the modifications to the file should be made at the same order in all the replicas. For this case, we propose the architecture shown in Figure 3(a-b) (for simplicity, we only discuss the single-replica case; similar results apply in the multi-replica case). This paradigm works as follows:

1. The NFS and the WB server are notified by the application at startup whether it wishes to be warmly backed-up (and which directories to back up).
2. When the NFS server receives a modify-request from a WB application, it must wait for a sequence-message from its warm-backup.
3. The WB server *snoops* for all NFS messages. When it receives an NFS modify-request from a WB application, it issues the modify-operation on the replica, and sends a numbered sequence-message with an identification of the request to the NFS server in the site of the accessed file.
4. The NFS server executes the modify-requests it has received according to the order set by the sequence-messages it receives from the WB server. It returns the results to the application.
5. If either the NFS server or the WB server loses a message from the application, the application will time-out and re-issue the request (this is the standard fault handling protocol of the NFS).

The modification of the NFS server can be done internally (Figure 3(a)), or by placing a special server on the NFS server's machine that mediates between the application and the NFS server (Figure 3(b)).

Recovery

The modified NFS server and the WB server need to dynamically detect each other's failures and recoveries, and bring the system to a consistent state upon recoveries. In case of the primary NFS server failure, the WB server holds up-to-date copies of all the accessed file. The WB can also "take over" the primary NFS server role for those files, and allow currently running applications to continue. In order that a running application will turn to the WB server for backup file-service, it needs to be modified as well. It is sufficient to transparently replace the system-calls library and no change is required to the application itself. The details of the *takeover* algorithm, for moving the application from the primary server to the backup, are standard for such a system, and are beyond the scope of this compact presentation. Likewise, the matter of re-integrating an NFS server upon recovery are detailed elsewhere ([15]).

Exclusive-Write WB

One of the drawbacks of the above architecture is that it requires changes to the NFS server, thus affecting the entire system and not only the WB applications. In this section we offer a more restrictive solution, that does not require changes to the NFS server. This solution work under the assumption that there are no concurrent accesses by different applications to the same file.

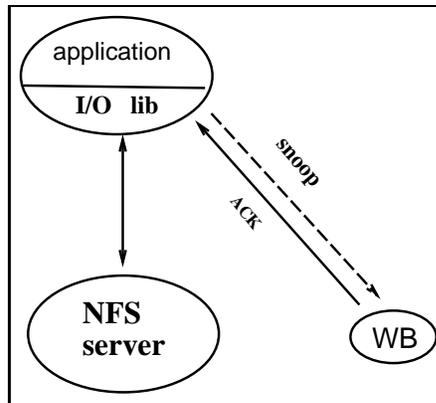


Fig. 4. Exclusive-Reader WB

It is a known belief that the majority of modify-accesses to files in Unix are done by processes exclusively and not concurrently. The Unix loose semantics on concurrent-modify on files encourages this style of usage. In order to replicate files for the exclusive writer case, we propose the architecture shown in Figure 4. Each application that wishes to obtain the WB services, links with a modified system-call library. The files accessed by this applications must be dedicated to

the WB server, and should be not accessed by “regular” NFS clients. The modify accesses to files is done as follows:

1. The WB server is notified by the application at startup whether it wishes to be warmly backed-up (and which directories to back up).
2. When the application issues a modify request on a file, it sends the request as usual to the NFS server.
3. There is no change to the NFS server: it executes each received request and returns the results to the application.
4. The WB server “snoops” on the network for NFS requests. When it receives a modify-request from a WB application, it performs the modify-operation on the replica, and returns an acknowledgment (ACK) message to the application.
5. The application waits for the returned results from the NFS server and for the acknowledgment message from the WB server. If it times out on any one of them, it re-issues the request. If either the NFS server or the WB server loses a message, it will receive the retransmission.

Practical Considerations

The WB architecture is designed for incurring a minimal overhead on the message traffic in the system. All the dashed-arrows in Figures 3, 4 are almost cost-free, and are done by network snooping (the reason for saying that this *almost* cost free is that in case the WB server loses a message, it needs to be re-transmitted). Furthermore, note that in all the proposed paradigms, the extra-messages employed by the WB system are very short messages that do not carry data (*e.g.* the sequence-message and the ACK message). Thus, for write-operations on files, the written data is sent only once over the network.

The common source of delay in all the proposed paradigms is the need to wait for an extra message from the WB server. We have implemented a prototype of the WB server over the Sun Network Interface Tap (NIT), and are currently experimenting with the performance of the system proposed in Figure 4.

The WB server snoops for messages addressed at multiple NFS servers. Therefore, it needs to put the network-interface in its machine in promiscuous mode, and filter the relevant messages among the multitude of messages transferred in the system. This requires the machine(s) that run the WB server to be fairly lightly-loaded. This indicates that for best results, the WB server should probably run on a designated machine by itself, the *backup machine*.

7 Conclusions

The hardware media of computer networks provide the capability to broadcast messages. This offers an efficient way to disseminate messages to multiple destinations. Essentially, this is a practical consideration; however, if we incorporate the broadcast capability into the system model, we arrive at distributed algorithms that are quite different from their sequential counterparts. Moreover, in

the case of the warm-replication application, the ability to snoop within a broadcast network has led us to devise a completely new scheme for replication. Thus, these practical considerations can be of significance to the designer of distributed services.

Future networks such as the high-speed FDDI ring, and wireless networks, also possess the broadcast capability. Therefore, understanding the potential in broadcast communication is important. Our experience with some of the protocols presented in this paper indicates that there are interesting tradeoffs that need to be explored. The choice between having a distributed control and a centralized control is not fully understood yet. Similarly, we note that *quantitative* measures may effect their conduct. For example, the reliable broadcast protocols we presented behave quite differently under various communication-load conditions, and when different loss-rates of underlying network messages are exhibited.

Randomized techniques have proved their importance in the field of distributed algorithms by producing solutions to the consensus problem and others. Rarely, is any of the theoretical randomized protocols used in practical distributed environments. Typically, this is because they are too complicated, or involve too many message-exchanges. We propose to investigate the usage of randomization in more realistic models, and in particular, within a broadcast domain.

8 Acknowledgments

The work presented in this paper benefitted from many other works. The Transis reliable broadcast and the membership protocol are the results of joint work with the colleague Transis developers, Yair Amir and Shlomo Kramer. The Warm Backup service was enhanced and implemented by Yuval Harari. Yuval Yarom helped editing the section about the Warm Backup. Ray Strong and Rudigue Reischuk co-developed with one of the authors the clock synchronization scheme. Idit Keidar wrote the tool that automatically plotted Figure 2.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LCNS, 647)*, number 6th, pages 292–312, November 1992.
2. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *Annual International Symposium on Fault-Tolerant Computing*, number 22, pages 76–84, July 1992.
3. Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Intl. conf. on Distributed Computing Systems*, 1993. to appear.
4. J. Auerbach, M. Gopal, M. Kaplan, and S. Kutten. Multicast group membership management in high speed wide area networks. In *proc. intl. conference on Distributed Computing Systems*, number 11, pages 231–238, May 1991.

5. K. Birman, R. Cooper, and B. Gleeson. Programming with Process Groups: Group and Multicast Semantics. TR 91-1185, dept. of Computer Science, Cornell University, Jan 1991.
6. K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual*. Dept of Computer Science, Cornell University, Sep 90.
7. K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.
8. D. R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Trans. Comput. Syst.*, 2(3):77–107, May 1985.
9. F. Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4(4), April 1991.
10. S. E. Deering. Host extensions for IP multicasting. RFC 1112, SRI Network Information Center, August 1989.
11. D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.
12. D. Dolev, S. Kramer, and D. Malki. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. In *Annual International Symposium on Fault-Tolerant Computing*, number 23, June 1993.
13. D. Dolev, R. Reischuk, and H.R. Strong. Clock Synchronization Algorithms on a LAN. in preparation, 1993.
14. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.
15. Y. Harari. Warm Backup Tool for Unix Network File System. internal manuscript, 1992.
16. F. Jahanian and W. Moran. Strong, Weak and Hybrid Group Membership. unpublished, IBM internal draft, 1992.
17. M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.
18. P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Trans. Parallel & Distributed Syst.*, (1), Jan 1990.
19. P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership Algorithms for Asynchronous Distributed Systems. In *Intl. Conf. Distributed Computing Systems*, May 91.
20. S. Mishra, L. L. Peterson, and R. D. Schlichting. A Membership Protocol based on Partial Order. In *proc. of the intl. working conf. on Dependable Computing for Critical Applications*, Feb 1991.
21. L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 89.
22. J. B. Postel. User Datagram Protocol. RFC 768, SRI Network Information Center, August 1980.
23. A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *proc. annual ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.
24. A. M. Ricciardi, K. P. Birman, and P. Stephenson. The Cost of Order in Asynchronous Systems. In *Intl. Workshop on Distributed Algorithms proceedings (WDAG-6)*, (LCNS, 647), number 6th, pages 329–345, November 1992.

This article was processed using the L^AT_EX macro package with LLNCS style