Voting with Ghosts

Robbert van Renesse
Andrew S. Tanenbaum

Dept. of Computer Science
Vrije Universiteit
The Netherlands

ABSTRACT

Data replication is a technique for increasing the availability of data. Two popular algorithms for maintaining consistency among the replicas are Weighted Voting [1] and Available Copies [2] In recent papers [3] it has been shown that under common circumstances Available Copies (AC) performs better than Weighted Voting (WV). However, the issue of network partitioning due to gateway crashes is ignored in AC. We present an improvement of WV that, if configured accordingly, performs as well as AC, but, unlike AC, also works correctly in the face of network partitioning.

1. INTRODUCTION

Data replication in distributed operating systems is a technique for increasing the availability of data. It can also increase the performance of the system, since an application can use nearby copies of the data instead of distant ones. A serious problem, however, is to make the collection of replicated data look like a single object, even under concurrent access. Users should always see the most recent version. This prerequisite is called *serial consistency* or *one-copy serializability*.

Two popular algorithms for achieving serial consistency are *Weighted Voting* [1] and *Available Copies* [2] Weighted Voting (WV) trades off read availability for write availability, and may need more than one copy available to perform an operation. Available Copies (AC) works as long as there is at least one available copy, but works incorrectly in the presence of communication errors or network partitioning.

We propose a method that has none of the disadvantages of the algorithms stated above. Both WV and AC are special cases of the proposed method, which we call *Voting with Ghosts* (VWG). The read and write availabilities in VWG are at least as good as those of WV and AC. VWG uses WV as its basic algorithm, but substitutes so-called *ghosts* for unavailable copies. The ghost may vote in the place of the original copy, but only for write quorums.

This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.) under grant 125-30-10.

We will assume that the network can only be partitioned at certain places (*e.g.*, gateways may fail). We will call a part of a network that cannot be further partitioned a *segment*. If a segment is down, the nodes within that segment will not be able to communicate with each other, nor with nodes on other segments. For example, a token ring could be broken preventing the token from being passed around. An Ethernet segment that is not terminated at one of its sides will not allow communication either. If a segment is up, all the running nodes in this segment can communicate among each other. We will need this property since, as we shall see, a ghost is started on the same segment as the unavailable copy. The assumption of a segmented network reflects current network technology [5].

In the following section we will describe the VWG method in detail. In section 3 we will analyze the method in the presence of node crashes and network partitions. Section 4 compares VWG to WV and AC. Section 5 draws some conclusions.

2. ALGORITHM

VWG uses WV as the basic algorithm. In section 2.1 we will briefly describe WV. In the following section we will introduce *ghosts* to increase the write availability of the basic algorithm significantly. The last section describes how ghosts are generated.

2.1. WEighted Voting

In WV every copy of a replicated object is assigned a number of votes. To read the object, the user has to collect a *read quorum* of r votes; to write the object a *write quorum* of w votes is needed. To ensure that there is an intersection between a read quorum and a write quorum, $r + w$ has to be greater than the total number of votes in all the copies, N, This intersection will always contain the most current version of the object, that is, the contents of the last write operation. To keep track of versions, a *version number* is associated with every object.

A write to a copy of the object contains the new version number of the object, which is the incremented current version number. If $w \neq N$ and the current version number is unknown, a read quorum is needed to obtain the current version number. If $w = N$ there is no need to maintain a version number since every copy will always contain the latest version. A user that reads the copies in a read quorum can get the latest version by taking the one with the highest version number. If a quorum cannot be obtained, the operation will either fail or wait until the quorum can be reached.

By having a small read quorum, and therefore a large write quorum, we obtain a high read availability, but a correspondingly low write availability. If one or more containers (= storage nodes) crash, and the write quorum can no longer be acquired, the object will not be available for writing.

2.2. Ghosts

It is this last problem that can be overcome with *ghosts*. In the event of a container crash, a ghost is started *within the same segment*. A ghost is a process without storage. It will be assigned the votes of the crashed object. How the crash of a container is detected and the ghost is started is the subject of the next section. For now we will assume that ghosts are started directly after a container crash.

Since the ghost has no storage, it cannot reply to a read request, and thus cannot participate in a read quorum. However, the ghost is allowed to participate in a write quorum. In this case it discards the data that is received with the subsequent write request, and it answers with a special reply. A write can only succeed if the write quorum contains at least one non-ghost copy. Basically, the voting algorithm uses ghosts to detect that a copy is unavailable due to a node crash, not due to a network partition.

When a container is rebooted, the copies in the container have to be brought up-to-date. Because although the ghost pretended that it performed the write operations, in reality it did not. A copy that is available again, but not yet restored, is said to be *comatose*. One strategy to make the copy available again is to keep acting like a ghost until a write request has been received.

Alternatively, a ghost can try to acquire a read quorum to copy the current version. If a read quorum is currently unavailable, the recovery will have to wait until either a read quorum is available, or *all* the nodes are either available or comatose. Now the latest version available among the copies of the objects is automatically the current version. Using this version the comatose objects can recover and can become available again.

Note that a write quorum must contain at least one stable container, that is, a container in which the copies survive crashes, and the contents and version number remain consistent with each other [6]. Otherwise it is possible that data be lost if all containers crash. WV, in contrast, requires that all containers are stable, since there is no recovery involved after a crash.

2.3. Boot Service

Container crashes are detected by the *boot service*. There is a boot service on each segment that has containers. All copies and their votes are registered with the boot service. It polls the containers at regular intervals, and expects an *I-am-ok* reply within a maximum time interval. If this reply is not received, the boot service will try to reboot the container, and poll the container again afterwards. If there is still no positive response, it will ensure that the container will stay unavailable. This can be done by resetting the hardware, disconnecting it from the network, or by switching off the power using an electronic switch. Now, in the same segment, it will start a ghost for the container. The boot server gives the votes of the container to the ghost, so that the container may vote instead of the original. If an application detects that a container is unavailable, it can speed up the crash detection process by sending a message to the boot service.

Ghosts crashes are handled in the same way. The boot service itself is protected against crashes by replicating it on several nodes in the segment. The boot service can use either WV or AC to maintain consistency internally. This consistency is required to get consensus about which containers are down, and which server is going to restart the container. Since the servers for the boot service reside on one segment they do not have to handle partitions. The boot servers can poll each other to recover from crashes. If the boot service becomes unavailable anyway, the algorithm degrades to WV.

2.4. Correctness

If the correctness of WV is understood, then the correctness of VWG is easily seen. A write quorum consists of w votes with at least one non-ghost copy. A read quorum consists of r votes with only non-ghost copies. Since $r + w > N$ we have a non-empty intersection between the read and the write quorum. Since the write quorum consists of current copies (the recovery always takes care of that), and the read quorum consists of non-ghosts, the intersection consists of current, non-ghost copies. It is a copy from this set that the read operation uses.

Serial consistency is guaranteed if reads and writes are atomic. This can be achieved by using a two-phase commit protocol [7]. Two-phase commit requires stable storage [6]. However, in a write operation, some (not all) participants can be ghosts, and they do not have storage at all. Fortunately, ghosts can still cooperate in an atomic action since they only *pretend* to execute the operation. They do so by always sending positive responses on messages from the coordinator of the two-phase commit protocol, and otherwise ignoring the contents of the messages.

3. ANALYSIS

In this section we will analyze the behavior of VWG in the event of container crashes and network partitions. We define *availability* as the mathematical probability of an object being accessible at any given time. Such an object could be replicated data as a whole, or just a single copy. An object could also be a gateway between segments of a network, or a segment itself. We will assume that the boot service is always available and creates ghosts immediately after a crash. This is not unreasonable for the analysis if the availabilities of the containers are reasonably high, such that ghost creation and recovery are not often.

Below we give a combinatorial analysis instead of stochastic analysis. A stochastic process model, using Markov chains, might give a better evaluation, but it relies on certain critical assumptions such as assuming exponential distribution of all events [8, 9]. This may be realistic for failure rates, but, for example, repair times have at least a large constant term representing the service call time. In Markov models, the probability of a partition is usually ignored. However, in our experience, wide-area networks are often partitioned. And even in a local internetwork the probability of a partition cannot be neglected when compared to the unavailability of a replica.

An alternative to stochastic process models is simulation, in which case the abovementioned simplifications need not be made [3, 5]. A disadvantage of simulation is that it does not provide algebraic expressions for availability of replicated objects, which are needed for general analysis of the protocols. With combinatorial analysis, as done in the next section, these expressions can be stated, although they are inaccurate if the availabilities of the components in the configuration are low. Since this is usually not the case, we believe that combinatorial analysis provides a reasonable insight in the performance of the replication protocols.

3.1. Node Crashes

For the time being we will assume that the network is not partitioned and is always available, and that each copy has the same (independent) availability p and has exactly one vote. We will calculate the read and write availability of data replicated using VWG. These calculations are based on the reliability theory of $k{-}out{-}of{-}N$ systems [10]. In section 4 we will compare the results with WV and AC.

$K{-}out{-}of{-}N$ gives the availability of replicated data if at least k of its N copies are needed for access. Clearly, $1{-}out{-}of{-}N$ is the probability that not all copies are inaccessible: $1-(1-p)^N$. Using probability theory we can see that:

$$k{-}out{-}of{-}N = 1 - \sum_{i=0}^{k-1} \binom{N}{i} p^i (1-p)^{N-i}$$

This formula gives the availability of data if k available copies are needed for a quorum. In VWG the read availability is $r{-}out{-}of{-}N$. Since a write operation only needs one available copy, the write availability of the data is $1{-}out{-}of{-}N$. Fig. gives the read and write availability of VWG for different combinations of r and w as function of the availability of the copies p. For these statistics hold $r + w = N + 1 = 9$. The dotted line in the graphs gives the availability of the data if it would not have been replicated. arrowht = 20; arrowwid = 20; A: [line from (0, 1000) to (0, 0) line to (1000, 0) "1 " at (-40, 1000) below "0" at (-40, -50) "1" at (1000, -50) rjust line dotted 30 from (1000, 1000) to (0, 0) arrow from (-100, 800) to (-100, 1000) arrow from (800, -100) to (1000, -100) $N = 8$ "avail" at (-100, 760) below "p" at (730, -100) ljust box ht 350 wid 1 invis "$r = 1$" with .s at (975, 0) move to (0, 0) line to (10, 77) line to (20, 149) $w = 8$ line to (30, 216) line to (40, 278) line to (50, 336) line to (60, 390) line to (70, 440) line to (80, 486) line to (90, 529) line to (100, 569) line to (110, 606) line to (120, 640) line to (130, 671) line to (140, 700) line to (150, 727) line to (160, 752) line to (170, 774) line to (180, 795) line to (190, 814) line to (200, 832) line to (210, 848) line to (220, 862) line to (240, 888) line to (250, 899) line to (270, 919) line to (280, 927) line to (300, 942) line to (310, 948) line to (330, 959) line to (340, 963) line to (350, 968) line to (360, 971) line to (370, 975) line to (380, 978) line to (390, 980) line to (400, 983) line to (410, 985) line to (430, 988) line to (440, 990) line to (450, 991) line to (510, 996) line to (520, 997) line to (530, 997) line to (550, 998) line to (560, 998) line to (580, 999) line to (590, 999) line to (1000, 1000) " RA = WA" at (140, 700) ljust] "(a)" at A.s below B: [line from (0, 1000) to (0, 0) line to (1000, 0) "1 " at (-40, 1000) below "0" at (-40, -50) "1" at (1000, -50) rjust line dotted 30 from (1000, 1000) to (0, 0) arrow from (-100, 800) to (-100, 1000) arrow from (800, -100) to (1000, -100) $N = 8$ "avail" at (-100, 760) below "p" at (730, -100) ljust box ht 350 wid 1 invis "$r = 3$" with .s at (975, 0) move to (0, 0) line to (30, 1) line to (40, 3) line to (60, 8) $w = 6$ line to (70, 14) line to (80, 21) line to (100, 38) line to (120, 60) line to (130, 74) line to (140, 89) line to (150, 105) line to (160, 122) line to (170, 141) line to (190, 181) line to (200, 203) line to (220, 248) line to (230, 272) line to (250, 321) line to (280, 397) line to (290, 422) line to (300, 448) line to (310, 473) line to (350, 572) line to (360, 595) line to (390, 663) line to (400, 684) line to (420, 724) line to (430, 744) line to (440, 762)

line to (450, 779) line to (470, 812) line to (480, 827) line to (490, 841)
line to (510, 868) line to (520, 880) line to (530, 891) line to (540, 901)
line to (560, 920) line to (570, 928) line to (590, 943) line to (610, 956)
line to (620, 961) line to (640, 970) line to (670, 981) line to (690, 986)
line to (730, 993) line to (780, 997) line to (790, 998) line to (800, 998)
line to (810, 999) line to (820, 999) line to (1000, 1000) " RA" at (408, 700)
ljust move to (0, 0) line to (10, 77) line to (20, 149) line to (30, 216)
line to (40, 278) line to (50, 336) line to (60, 390) line to (70, 440)
line to (80, 486) line to (90, 529) line to (100, 569) line to (110, 606)
line to (120, 640) line to (130, 671) line to (140, 700) line to (150, 727)
line to (160, 752) line to (170, 774) line to (180, 795) line to (190, 814)
line to (200, 832) line to (210, 848) line to (220, 862) line to (240, 888)
line to (250, 899) line to (270, 919) line to (280, 927) line to (300, 942)
line to (310, 948) line to (330, 959) line to (340, 963) line to (350, 968)
line to (360, 971) line to (370, 975) line to (380, 978) line to (390, 980)
line to (400, 983) line to (410, 985) line to (430, 988) line to (440, 990)
line to (450, 991) line to (510, 996) line to (520, 997) line to (530, 997)
line to (550, 998) line to (560, 998) line to (580, 999) line to (590, 999)
line to (1000, 1000) " WA" at (140, 700) ljust] with .n at A.s - (0, 300) "(b)" at B.s below

The read availability (RA) and write availability (WA) of a replicated object under VWG as function of the availability of the copies, for different values of *r* and *w*. The dotted lines give the availability if the object would not have been replicated.

3.2. Network Partitioning

In this section we will calculate the read and write availability in VWG from any part of the network. We will allow network partitions, and that copies have different availabilities and numbers of votes. Note that the availability of a replicated object is different observed from different locations in the network. First we will represent a network as a graph. We represent containers, segments, and gateways between segments as nodes in the graph, and the connections between these parts of the network as edges. Each node in the graph has an availability and a number of votes attached to it. Segment nodes always have 0 votes, but a gateway may well have votes if the machine is also used for storage.

An example of a network and the corresponding graph is shown in Fig. . Here we have a backbone network (1) that connects four segments (4-7) using two repeaters (2, 3). An extra gateway (11) is added between segments 5 and 6. Nodes 8 to 14, including gateway 11, are containers. In the graph representation we see all the parts of this configuration as nodes. For ease of use we made the segment nodes elliptical and the gateway nodes square. boxht = 50; boxwid = 50; circlerad = 30 moveht = 150; movewid = 120; linewid = 100 arrowht = 30; arrowwid = 30 [down X: [right; line right A: box; move; B: circle; move; C: circle; move; D: circle "2" at A.c - (0,3) "8" at B.c - (0,3) "9" at C.c - (0,3) "10" at D.c - (0,3) line from A.n up 50; line "4" above right 600 line from A.s down 50; line "5" below right 600 line from B.n up 45 line from C.n up 45 line from D.s down 45 E: line from 1/2 <C, D> - (0, 75) down 50] Y: box wid 80 with .n at X.E.end "11" at Y.c - (0,3) Z: [right; line right A: box; move B: circle; move C: circle; move D: circle "3" at A.c -

(0,3) "12" at B.c - (0,3) "13" at C.c - (0,3) "14" at D.c - (0,3) line from A.n up 50; line "6" above right 600 line from A.s down 50; line "7" below right 600 line from B.n up 45 line from C.n up 45 line from D.s down 45 E: line from 1/2 <C, D> + (0, 75) up 50] with last [].E.end at Y.s line " 1" ljust from X.w + (0, 100) to Z.w + (0, -100)] move; arrow; move [A: ellipse ht 90 wid 40 B: box at A + (200, 80) C: box at A + (200, -80) D: ellipse ht 40 wid 90 at B + (200, 160) E: ellipse ht 40 wid 90 at B + (200, 0) F: ellipse ht 40 wid 90 at C + (200, 0) G: ellipse ht 40 wid 90 at C + (200, -160) H: circle at D + (200, 40) I: circle at D + (200, -40) J: circle at E + (200, 40) K: box wid 80 at A + (600, 0) L: circle at F + (200, -40) M: circle at G + (200, 40) N: circle at G + (200, -40) "1" at A.c - (0,3) "2" at B.c - (0,3) "3" at C.c - (0,3) "4" at D.c - (0,3) "5" at E.c - (0,3) "6" at F.c - (0,3) "7" at G.c - (0,3) "8" at H.c - (0,3) "9" at I.c - (0,3) "10" at J.c - (0,3) "11" at K.c - (0,3) "12" at L.c - (0,3) "13" at M.c - (0,3) "14" at N.c - (0,3) line from A.ne to B.w; line from A.se to C.w line from B.n to D.sw; line from B.e to E.w line from C.e to F.w; line from C.s to G.nw line from D.ne to H chop 0 chop 30 line from D.se to I chop 0 chop 30 line from E.ne to J chop 0 chop 30 line from E.se to 1/3 <K.nw, K.sw> line from F.ne to 2/3 <K.nw, K.sw> line from F.e to L chop 0 chop 30 line from F.se to M chop 0 chop 30 line from G.se to N chop 0 chop 30] A typical segmented network and its graph representation. The square nodes are gateways, and the elliptical nodes are network segments.

A simple algorithm for calculating the availability from any node in the graph can be constructed as follows. The algorithm considers every possible state of the segmented network. For each state (up/down combination of all the nodes) it calculates the probability of this state. This is the product of the availabilities of the nodes that are up, and the *unavailabilities* (one minus the availability) of the nodes that are down. From the node that we are interested in a check is made whether or not enough votes can be acquired by traveling the graph, stopping at unavailable nodes. If a quorum can be reached, the probability of the combination is added to the total availability. The complete algorithm can be found in Appendix A.

Using the algorithm on the example of Fig. , we find the availabilities in Fig. for quorums of 1, 5, and 16 votes. Users on segment 7 have, for a read quorum of 5, a relatively low availability. To improve this, we could try to move a container to segment 5, or to increase the number of votes of node 14. For comparison, the last column contains the availabilities if the data was not replicated, and situated at node 13. (This column was generated by assigning all votes to node 13, and none to the other nodes.) For example, a user on segment 5 needing a quorum of 1 vote finds an availability of the data of 0.970. If the data would not have been replicated, but stored only on node 13, the availability would have been 0.912. Moreover, in the unlucky event that node 13 crashes, the data would be completely unavailable.

input			*output*			
node number	# votes	avail	quorum 1	quorum 5	quorum 16	no repl. data on 13
1	0	.98	.977	.976	.504	.918
2	0	.95	.950	.950	.514	.891
3	0	.95	.950	.948	.514	.894
4	0	.97	.970	.965	.514	.865
5	0	.97	.970	.969	.514	.912
6	0	.97	.970	.969	.514	.941
7	0	.97	.962	.919	.514	.867
8	1	.93	.930	.901	.514	.804
9	4	.98	.980	.947	.514	.847
10	1	.94	.940	.911	.514	.857
11	4	.99	.990	.989	.514	.931
12	1	.92	.920	.892	.514	.866
13	3	.97	.970	.940	.514	.970
14	2	.85	.850	.782	.514	.737

The availabilities for different quorums of the example network.

4. COMPARISON

WV and AC are two popular algorithms for maintaining serial consistency of replicated data. In this section we will compare the availability of replicated data in these algorithms to VWG. For the probabilistic analysis we will assume again a network without partitions, that each copy has one vote, and that each copy has the same (independent) availability p.

4.1. Weighted Voting

The pure WV algorithm has been described in section 2.1. The read availability of the data is $r-out-of-N$, which is the same as that of VWG. The write availability is $w-out-of-N$. If we want high read availability, we have to choose a small read quorum. Since $r + w > N$ we will have a large write quorum, and thus low write availability. The read and write availability for different combinations of r and w are shown in Fig. . The contrast with VWG can be observed by comparing Fig. to Fig. 1. arrowht = 20; arrowwid = 20; A: [line from (0, 1000) to (0, 0) line to (1000, 0) "1 " at (-40, 1000) below "0" at (-40, -50) "1" at (1000, -50) rjust line dotted 30 from (1000, 1000) to (0, 0) arrow from (-100, 800) to (-100, 1000) arrow from (800, -100) to (1000, -100) $N = 8$ "avail" at (-100, 760) below "p" at (740, -100) ljust box ht 350 wid 1 invis "$r = 1$" with .s at (975, 0) move to (0, 0) line to (10, 77) line to (20, 149) line to (30, 216) $w = 8$ line to (40, 278) line to (50, 336) line to (60, 390) line to (70, 440) line to (80, 486) line to (90, 529) line to (100, 569) line to (110, 606) line to (120, 640) line to (130, 671) line to (140, 700) line to (150, 727) line to (160, 752) line to (170, 774) line to (180, 795) line to (190, 814) line to (200, 832) line to (210, 848) line to (220, 862) line to (240, 888) line to (250, 899) line to (270, 919) line to (280, 927) line to (300, 942)

line to (310, 948) line to (330, 959) line to (340, 963) line to (350, 968)
line to (360, 971) line to (370, 975) line to (380, 978) line to (390, 980)
line to (400, 983) line to (410, 985) line to (430, 988) line to (440, 990)
line to (450, 991) line to (510, 996) line to (520, 997) line to (530, 997)
line to (550, 998) line to (560, 998) line to (580, 999) line to (590, 999)
line to (1000, 1000) " RA" at (140, 700) ljust move to (0, 0) line to (430, 1)
line to (440, 1) line to (460, 2) line to (470, 2) line to (490, 3) line to (500, 3) line to (510, 4) line to (570, 11) line to (580, 12) line to (590, 14)
line to (610, 19) line to (620, 21) line to (630, 24) line to (640, 28)
line to (650, 31) line to (660, 36) line to (670, 40) line to (680, 45)
line to (690, 51) line to (710, 64) line to (720, 72) line to (740, 89)
line to (750, 100) line to (770, 123) line to (780, 137) line to (800, 167)
line to (810, 185) line to (820, 204) line to (830, 225) line to (840, 247)
line to (850, 272) line to (860, 299) line to (870, 328) line to (880, 359)
line to (890, 393) line to (900, 430) line to (910, 470) line to (920, 513)
line to (930, 559) line to (940, 609) line to (950, 663) line to (960, 721)
line to (970, 783) line to (980, 850) line to (990, 922) line to (1000, 1000)
" WA" at (957, 700) ljust] "(a)" at A.s below B: [line from (0, 1000) to (0, 0) line to (1000, 0) "1 " at (-40, 1000) below "0" at (-40, -50) "1" at (1000, -50) rjust
line dotted 30 from (1000, 1000) to (0, 0) arrow from (-100, 800) to (-100, 1000)
arrow from (800, -100) to (1000, -100) $N = 8$ "avail" at (-100, 760) below "p" at (740, -100) ljust box ht 350 wid 1 invis "$r = 3$" with .s at (975, 0) move to (0, 0) line to (30, 1) line to (40, 3) line to (60, 9) $w = 0$ line to (70, 14) line to (80, 21) line to (100, 38) line to (120, 60) line to (130, 74) line to (140, 89) line to (150, 105) line to (160, 122) line to (170, 141) line to (190, 181) line to (200, 203)
line to (220, 248) line to (230, 272) line to (250, 321) line to (280, 397)
line to (290, 422) line to (300, 448) line to (310, 473) line to (350, 572)
line to (360, 595) line to (390, 663) line to (400, 684) line to (420, 724)
line to (430, 744) line to (440, 762) line to (450, 779) line to (470, 812)
line to (480, 827) line to (490, 841) line to (510, 868) line to (520, 880)
line to (530, 891) line to (540, 901) line to (560, 920) line to (570, 928)
line to (590, 943) line to (610, 956) line to (620, 961) line to (640, 970)
line to (670, 981) line to (690, 986) line to (730, 993) line to (780, 997)
line to (790, 998) line to (800, 998) line to (810, 999) line to (820, 999)
line to (1000, 1000) " RA" at (408, 700) ljust move to (0, 0) line to (200, 1)
line to (210, 1) line to (220, 2) line to (230, 2) line to (240, 3) line to (290, 9) line to (330, 18) line to (350, 25) line to (380, 38) line to (400, 49)
line to (410, 56) line to (430, 71) line to (450, 88) line to (460, 98)
line to (480, 119) line to (490, 131) line to (500, 144) line to (510, 158)
line to (530, 187) line to (540, 203) line to (550, 220) line to (570, 255)
line to (580, 275) line to (590, 294) line to (600, 315) line to (620, 358)
line to (630, 381) line to (660, 451) line to (670, 476) line to (710, 577)
line to (720, 602) line to (730, 628) line to (740, 653) line to (770, 727)
line to (790, 774) line to (800, 796) line to (820, 839) line to (830, 858)
line to (850, 894) line to (860, 910) line to (870, 925) line to (880, 939)
line to (890, 951) line to (900, 961) line to (920, 978) line to (940, 990)
line to (950, 994) line to (960, 996) line to (980, 999) line to (990, 999)

line to (1000, 1000) " WA" at (759, 700) ljust] with .n at A.s - (0, 300) "(b)" at B.s below The read availability (RA) and write availability (WA) of a replicated object under WV as function of the availability of the copies, for different values of *r* and *w*.

VWG gives for any combination of *r* and *w* optimal write availability. In a non-partitionable network we get optimal overall availability by choosing $r = 1$ and $w = N$. A disadvantage of VWG is that copies need to be restored after crashes. This involves disallowing write access to one copy during the recovery. If crashes are sufficiently rare, this disadvantage need not be a problem.

To allow caching of data, *weak representatives* [1] can be used. Weak representatives are temporary copies that do not have any votes, but can still be included in any quorum. Especially a user can include a weak representative on the local machine in read quorums. If the weak representative has the current version number, the data can be copied from that representative, instead of from a remote node. In VWG it is unnecessary to create ghosts for weak representatives; after a crash the weak representative disappears completely.

To cut down on the storage requirements *witnesses* [11] can be used. Witnesses are copies that only maintain the version number of the copy, and not the data themselves. In a quorum there has to be at least one real copy. Otherwise witnesses have no effect on the voting mechanism. A concept similar to a ghost is a *temporary witness*. A temporary witness is dynamically substituted for a node when an *obsolete copy* (a copy that does not hold the current version) is being written. A ghost, on the other hand, is substituted for a *crashed copy*, and does not maintain the version number of the original at all.

4.2. Available Copies

AC reads one copy, and writes all available copies. When an unavailable copy becomes available again it is recovered from any available copy. If no copy is available, the recovery procedure will have to wait until all sites are comatose (rebooted but not restored), and then select the most recent version, as is done in VWG. An improvement of this strategy can be achieved by keeping track of which sites were available in every update, so recovery can take place with only the sites that were available at the last update. (This technique can be applied to VWG as well.)

AC does not tolerate communication errors. If, due to a bad connection, an available copy appears to be unavailable, the copy will become inconsistent. In the case of a network partition the algorithm will allow writes in all parts, thereby creating inconsistent versions of the data. arrowht = 20; arrowwid = 20; line from (0, 1000) to (0, 0) line to (1000, 0) "1 " at (-40, 1000) below "0" at (-40, -50) "1" at (1000, -50) rjust line dotted 30 from (1000, 1000) to (0, 0) arrow from (-100, 800) to (-100, 1000) arrow from (800, -100) to (1000, -100) "avail" at (-100, 760) below "*p*" at (730, -100) ljust move to (0, 0) line to (10, 77) line to (20, 149) line to (30, 216) line to (40, 278) line to (50, 336) line to (60, 390) line to (70, 440) line to (80, 486) line to (90, 529) line to (100, 569) line to (110, 606) line to (120, 640) line to (130, 671) line to (140, 700) line to (150, 727) line to (160, 752) line to (170, 774) line to (180, 795) line to (190, 814) line to (200, 832) line to (210, 848) line to (220, 862) line to (240, 888) line to (250, 899) line to (270, 919) line to (280, 927) line to (300, 942) line to (310, 948) line to (330, 959) line to (340, 963) line to (350, 968) line to (360, 971) line to (370, 975) line to (380, 978) line to (390, 980) line to (400, 983) line to (410, 985) line to (430, 988) line to (440, 990) line to (450, 991) line to (510, 996) line to (520, 997) line to (530, 997) line to (550, 998) line to (560, 998) line to (580, 999) line to (590, 999) line to (1000, 1000) " RA = WA" at (140, 700) ljust The availability (read and write) of AC.

To compare AC with VWG we will assume perfect communication (no errors, no partitioning). The availability of replicated data using the AC method is given in Fig. . For both read and write availability only one copy is needed. If we compare Fig. to Fig. 1 (a) we see that VWG, with a read quorum of 1, performs the same as AC. However, VWG also works correctly with non-perfect communication, and then allows flexible configuration for read and write availability. Furthermore, VWG needs to lock only one copy for recovery, instead of all.

An interesting alternative to AC is Regeneration [12]. Here unavailable copies are immediately replaced by new copies on other containers (provided that other containers are available). Using this technique the availability of data can be restored after container crashes. This technique can also be used in VWG. After a crash detection, the boot service can first try to generate a new copy on a different container within the same segment. If this fails, it can start a ghost as usual.

5. CONCLUSION

Voting with Ghosts combines the advantages of Weighted Voting and the advantages of Available Copies. It provides the flexibility of configuration as in WV, and the high write availability of AC. We have presented a technique for calculating the availability of replicated data from any point in an internetwork, to allow designing an optimal configuration.

Using VWG we are currently designing a replicated directory service for the Amoeba distributed operating system [13]. A directory maps ASCII names to object identifiers in the form of capabilities [14]. We will support atomic lookup, install, and delete operations on directory entries of multiple directories. In a version-based system this will allow installing a new version of a set of objects atomically. Amoeba already provides a boot service to support immortality of services.

6. ACKNOWLEDGEMENTS

We thank Erik Baalbergen, Henri Bal, Jehan-Fran,cois Pˆaris, Calton Pu, Greg Sharp, and Jennifer Steiner for the careful reading of our paper.

7. REFERENCES

References

1. D. K. Gifford, ''Weighted Voting for Replicated Data,'' *Proc. of the 7th Symp. on Operating System Principles*, pp. 150-162, Pacific Grove, CA, USA, December 1979.
2. P. A. Bernstein and N. Goodman, ''An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases,'' *ACM Trans. on Database Systems*, vol. 9, no. 4, pp. 596-615, December 1984.
3. J. D. Noe and A. Andreassian, ''Effectiveness of Replication in Distributed Computer Networks,'' *Proc. of the 7th Int. Conf. on Distr. Computing Systems*, pp. 508-513, West Berlin, September 1987.
4. J. L. Carroll, D. D. E. Long, and J.-F. Pˆaris, ''Block-Level Consistency of Replicated Files,'' *Proc. of the 7th Int. Conf. on Distr. Computing Systems*, pp. 146-153, West

Berlin, September 1987.

5. J.-F. Pâris and D. D. E. Long, ''Efficient Dynamic Voting Algorithms,'' *Proc. of the 4th Int. Conf. on Data Engineering*, Los Angeles, CA, February 1988.

6. B. W. Lampson, ''Atomic Transactions,'' in *Distributed Systems — Architecture and Implementation*, pp. 246-265, Springer Verlag, West Berlin, 1981.

7. J. N. Gray, ''Notes on Data Base Operating Systems,'' in *Operating Systems: an Advanced Course*, ed. G. Goos and J. Hartmanis, Springer-Verlag, 1978.

8. D. D. E. Long and J.-F. Pâris, ''On Improving the Availability of Replicated Files,'' *Proc. of the 6th Symp. on Reliability in Distr. Software and Database Systems*, pp. 77-83, Kingsmill-Williamsburg, VA, March 1987.

9. M. Ahamad and M. H. Ammar, ''Performance Characteristics of Quorum-Consensus Algorithms for Replicated Data,'' *Proc. of the 6th Symp. on Reliability in Distr. Software and Database Systems*, pp. 161-168, Kingsmill-Williamsburg, VA, March 1987.

10. R. E. Barlow and K. D. Heidtmann, ''Computing k-out-of-N Reliability,'' *IEEE Trans. on Reliability*, vol. R-33, no. 4, pp. 322-323, October 1984.

11. J.-F. Pâris, ''Voting with Witnesses: A Consistency Scheme for Replicated Files,'' *Proc. of the 6th Int. Conf. on Distr. Computing Systems*, pp. 606-620, Cambridge, MA, May 1986.

12. C. Pu, J. D. Noe, and A. Proudfoot, ''Regeneration of Replicated Objects: a Technique and Its Eden Implementation,'' *Proc. of the Int. Conf. on Data Engineering*, pp. 175-186, February 1986.

13. S. J. Mullender and A. S. Tanenbaum, ''The Design of a Capability-Based Distributed Operating System,'' *The Computer Journal*, vol. 29, no. 4, pp. 289-300, March 1986.

14. A. S. Tanenbaum, S. J. Mullender, and R. van Renesse, ''Using Sparse Capabilities in a Distributed Operating System,'' *Proc. of the 6th Int. Conf. on Distr. Computing Systems*, pp. 558-563, Cambridge, MA, May 1986.

Appendix A

```
(* This algorithm calculates the availability of replicated data in a network of vulnerable    *)
(* components with independent availabilities.  The availability appears different from node   *)
(* to node.  The network can be described using a connection matrix of type bmatrix, where     *)
(* bmatrix[i][j] is true if there is a connection between nodes i and j.  bmatrix[i][j] is      *)
(* false if i = j or if there is no connection.                                                  *)

const NNODES = ...;    (* number of nodes in network graph *)
type  nodenum = 1..NNODES;
      bvector = array[nodenum] of boolean;  ivector = array[nodenum] of integer;
      rvector = array[nodenum] of real;     bmatrix = array[nodenum] of bvector;

(* Calculate availability of data in a node of a network graph *)
function analyze(
      connected: bmatrix; (* graph description *)
      votes:     ivector; (* votes per node *)
      avail:     rvector; (* availability of nodes *)
      node:      nodenum; (* node to investigate *)
      quorum:    integer  (* quorum to acquire *)     ): real;
   var up, visited: bvector; prob, result: real;
       i: nodenum; done: boolean;

(* Recursive function to walk the graph starting in node ''src.'' Returns the number of votes  *)
(* that it could collect. Pruning is done if the quorum is reached before searching all nodes. *)
   function collect(src: nodenum; have: integer): integer;
      var dst: nodenum;
      begin     (* collect *)
         if up[src] then           (* only visit nodes that are up *)
         begin
            visited[src] := TRUE;
            have := have + votes[src];       (* acquired some votes *)
            (* recursively visit all nodes, prune if have >= quorum *)
            dst := 1;
            while (have < quorum) and (dst <= NNODES) do
            begin
               if not visited[dst] and connected[src][dst] then
                  have := collect(dst, have);
               dst := dst + 1
            end
         end;
         collect := have
      end;      (* collect *)

   begin        (* analyze *)
      for i := 1 to NNODES do up[i] := FALSE;
      result := 0;
      repeat    (* for each up/down combination of nodes *)
         (* calculate probability of this up/down combination *)
         prob := 1.0;
         for i := 1 to NNODES do
         begin
            if up[i] then prob := prob * avail[i]
                     else prob := prob * (1 - avail[i]);
            visited[i] := FALSE
         end;
         (* if quorum collected add probability to result *)
```

```
    if collect(node, 0) >= quorum then result := result + prob;
    (* calculate next up/down combination *)
    i := 1;
    repeat
      up[i] := not up[i];
      if up[i] or (i = NNODES) then done := TRUE
      else begin done := FALSE; i := i + 1 end
    until done
  until not up[i];
  analyze := result
end;        (* analyze *)
```