ELSEVIER

PII: S0951-8320(96)00012-9

Reliability Engineering and System Safety **52** (1996) 139–151

0951-8320/96/$15.00

A taxonomy of voting schemes for data fusion and dependable computation

Behrooz Parhami

Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106-9560, USA

(Received 29 July 1995; accepted 24 January 1996)

Voting is an important operation in the fusion of data originating from diverse sources and in the realization of ultrareliable systems based on multiple computation channels. Voting involves the derivation of an output data object from a collection of n input data objects, as prescribed by the requirements and constraints of a voting algorithm. The objects voted on can be quite complex in terms of content and, explicitly specified or implicit, structure. Regardless of implementation details (e.g., whether realized in hardware, software, or hybrid schemes) and object space properties, voting algorithms can be classified according to how they view the input and output data objects and how they handle the votes (weights) at input and output. A 16-class (binary 4-cube) categorization results from dichotomizing each of the above voter features. This categorization leads to an abstraction that helps in the study of voting algorithms with regard to the dependability level for the outputs and the speed at which they are obtained; viz, the quality and efficiency of the algorithms. The taxonomy is broad enough to cover, and detailed enough to distinguish among, a wide variety of commonly used voting algorithms in data fusion and dependable computation. It also provides insight into the relationships of various voting schemes and facilitates comparison and fine-tuning of such algorithms. © 1996 Elsevier Science Limited.

1 INTRODUCTION

Voting is an important operation in the fusion of data originating from multiple sources and in the realization of ultrareliable systems that are based on the multi-channel computation paradigm. In data fusion, voting is a possible way of combining diverse data provided by multiple sources (such as sensors) whose outputs may be erroneous, incomplete, tardy, or totally missing. In ultrareliable systems, voting is required whether the multiple computation channels consist of redundant hardware units, diverse program modules executed on the same basic hardware, identical hardware and software with diverse data, or any other combination of hardware/program/data redundancy and/or diversity.

Depending on data volume and voting frequency, hardware or software voting schemes may be appropriate. Low-level, frequent voting requires the use of hardware voters whereas high-level voting on the results of fairly complex computations can be performed in software without serious performance degradation or overhead. However, the hardware/software distinction is superficial: it is the voting algorithm, and not the implementation scheme, that is important.

Even though voting algorithms have been studied extensively in the context of particular applications and for the realization of specific systems, a general investigation of the various design options and tradeoffs has not been attempted. There are signs, however, that research in this area is moving towards higher-level, application-independent (and, thus, more widely applicable) techniques. A general study of voting must deal with several concerns:

- minimizing the hardware cost or software overhead associated with voting,
- achieving high throughput and/or high speed (low latency) in voting,
- providing the flexibility of unequal/adaptive votes for different channels,
- utilizing the information coming from multiple channels in an optimal way,
- identifying the strengths and weaknesses of proposed voting methods,

- quantifying the correctness probability for the results of voting.

The significance of low-cost or low-overhead designs is obvious. The voting delay or latency, which is particularly important in real-time systems with hard deadlines, is also significant in high-performance systems. When data to be voted on is generated at a high rate, the voter must be able to keep up with the processing speed. In some such cases, the actual voting delay may not be critical but the voter throughput must match or exceed the input data rate. Unequal or adaptive vote weights are required in order to accommodate a priori or acquired knowledge about the relative trustworthiness of various data and computational resources. With respect to the optimal utilization of information, the voting scheme should be neither too optimistic nor overly pessimistic. Studying the strengths and weaknesses of proposed voting schemes would allow more informed selection of algorithms for each application context. Finally, due to algorithmic limitations or implementation flaws, no voting scheme is perfect and it is thus important to quantify the risks associated with any proposed approach.

The above aspects are interrelated and collectively challenge the designers with interesting reliability, algorithm complexity, and performance/overhead tradeoff issues. Many of these issues are being addressed by researchers in data fusion and dependable computation.[53] In order to study these aspects in a systematic way, a unified high-level view of voting is essential. Hence, our motivation is to propose a taxonomy that would expose the similarities and differences of proposed voting schemes.

The rest of this paper is organized as follows. Section 2 contains a review of voting methods and their applications. Section 3 defines voting in a much more general way than is commonly done, thus serving to unify diverse views. Section 4 contains the essence of the proposed taxonomy; viz the four dichotomies (exact/inexact, consensus/mediation, oblivious/adaptive, and threshold/plurality) as well as the resulting 16 classes. Sections 5, 6, 7 and 8 contain more detailed discussions of the four dichotomies mentioned above. Significance of the proposed taxonomy and directions for further research are discussed in the concluding Section 9.

2 VOTING AND ITS APPLICATIONS

The use of voting for obtaining highly reliable data from multiple unreliable versions was first suggested in the mid 1950s.[69] Since then, the concept has been practically utilized in fault-tolerant computer systems and has been extended and refined in many different ways. Reliability modeling of voting schemes by considering compensating errors,[60] handling of imprecise or approximate data, [14] combination with standby or active redundancy,[38] reconfigurable voting with declining replication factor upon detected failures,[39] voting on digital 'signatures' obtained from computation states in order to reduce the amount of information to be voted on,[63] and dynamic modification of vote weights based on a priori reliability data[56] constitute some of these extensions and refinements.

More recently, generalized voting with unequal input votes has been suggested for maintaining the reliability and consistency of data stored with replication in distributed computer systems.[20] This has become a very active research area. Voting is also used in the fusion of data obtained from multiple sensors.[23,24,36,37,65] In particular, when sensor outputs are abstracted as intervals, the voting scheme used is a type of approval voting.[52] However, the process used for multi-sensor data fusion is not commonly characterized as voting, which is often equated with majority agreement and other simple decision schemes.

Hardware voters that have been described in the literature are essentially 'bit-voters' that compute a majority function on n input bits.[26,62] Combined bit voting and disagreement detection has also been discussed.[15] Hardware voting on words and higher-level data objects has traditionally been handled by using independent parallel bit-voters or feeding the data sequentially through a single unit. Such independent bit-voting yields results that are optimistic, particularly when correlated errors are likely. Several algorithms and design techniques for hardware voters with adjustable or variable vote weights have been published.[45,47,48] The above hardware voting schemes all deal with voting on exact values (typically bit strings representing logical decisions or integer numerical values). Approximate or other context-dependent voting algorithms have not been implemented in hardware.

Hardware voters used in actual systems include 3-way voters in MIT's FTMP design,[22] Carnegie-Mellon University's C.vmp system,[61] and August Systems' industrial control computers.[71] The Jet Propulsion Laboratory's STAR computer[4] used a special 2-out-of-5 voter that was symmetrically connected to 3 active modules and 2 standby spares for its critical Test and Repair Processor (aerospace computers utilized hardware voters even before STAR). The effect of switch complexity on reliability and the design of low-complexity, and thus highly reliable, switch-voters for hybrid redundancy was considered by Siewiorek and McCluskey.[58,59] This work was instrumental in attracting attention to the

importance of simple switch-voters and led to several other designs including the self-purging[35] and sift-out[16] variations of adaptive voting with vote weights in $\{0, 1\}$.

Proposed software voters are quite varied and possess a wide range of features. The earliest software voters are found in the design of modular multiprocessors with replicated software. For example, the voter routine in SRI International's SIFT design[70] is invoked by any task which requires inputs for a new iteration. It uses tables provided by the local reconfiguration task to determine which processors contain copies of the required output (and in which of their buffers), reads the data from the appropriate buffers and uses a majority rule to obtain a single value. In the Space Shuttle's 4-way software voting scheme,[63] selected data items are computationally combined to form 'compare words' that are periodically exchanged and compared in 4 out of the 5 on-board computers. The 'stepwise negotiating voting' scheme[27] essentially amounts to a 2-out-of-n threshold voting strategy. The advantages of such 'relaxed' (non-majority) voting schemes have been discussed by others as well.[1,55]

Researchers in the field of software diversity have designed voters that are suitable for processing the results obtained by multiple versions of a program and have contributed techniques for handling approximate results.[5,11,29,33,68] Similar considerations would apply to voting schemes with data diversity.[3] Software voters have also been designed in connection with the management of replicated data in distributed systems[6,7,18,20,25,64,67] to assure database reliability and/or consistency. A common way to achieve this goal is to assign votes to participating nodes in the distributed system and to implement mutual exclusion by requiring each operation agent to 'collect' a certain number of consenting votes. Typical issues dealt with in this area are vote assignments to maximize reliability or to minimize average transaction response time (e.g., by adjusting 'read' and 'write' quorums). The delays resulting from synchronized voting in real-time systems and the expected time to collect a prescribed number of votes from distributed sites with different response characteristics have been studied in Refs. 57 and 42, respectively.

Research on voting algorithms has dealt with both implementation and effectiveness of various voting schemes. In the area of algorithm implementation, the main focus has been simple majority voting, which is a special case of the problem of finding repeated elements in a set.[10,13,21,41] In another line of attack,[34] the notion of 'approximate voting', henceforth applied only to real-valued numerical results, has been generalized by demonstrating that various forms of inexact voting can be easily applied to any metric space or, in the case of weighted averaging scheme, to any real vector space. As for effectiveness, Ref. 34 contains a qualitative comparison of several voting schemes under different error conditions. The problem of selecting the best voting scheme in order to maximize the probability of obtaining a correct result has been investigated in Refs. 8, 9, 19 and 40. Voting algorithms have also been studied from the viewpoint of sequential and parallel computational complexity.[32,46,49,52]

Finally, several researchers have presented generalized or mixed-mode voting schemes based on extending the voting domain, the fault model, or assumptions about the data redundancy scheme used.[2,28,31] Clearly, such useful generalizations must be covered in any systematic study of voting or proposed taxonomy.

3 A GENERAL VIEW OF VOTING

In order to facilitate and systematize the study of voting schemes, we have previously categorized them according to implementation in hardware or software (voting *networks*[45] or voting *routines*) and based on the size and structure of the input object space (see Fig. 1). A voting *algorithm*[52] specifies how the voting result is obtained from the input data and may be the basis for a voting network or a voting routine. For brevity, the term 'voter' is sometimes used as an equivalent for 'voting network' or 'voting routine', resulting in terms such as 'bit-voter', 'word-voter', or 'inexact majority voter'.

As shown in Fig. 1, the input objects to be voted upon can be atomic or composite. Composite objects, consisting of structured collections of atomic objects, have not received due attention in previous works on voting. With atomic objects, the input object space can be small or large. For small object spaces, further classification is unimportant, as they support very simple and efficient voting algorithms. For large object spaces, whether or not a distance metric can be defined, and as a special case, if the objects can be ordered, is important. Finally, for an unordered space,

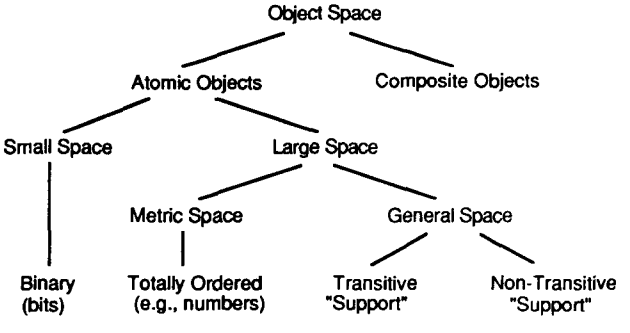

Fig. 1. Voting schemes classified according to the object space size and structure.

voting algorithms tend to be less complex if the notion of 'support', as discussed in Definition 3.1, is transitive (X supports Y and Y supports Z implies X supports Z).

Uses of voting in dependable multi-channel computation are almost exclusively based on atomic objects (primarily bits and numeric words), but data fusion routinely involves the processing of composite objects. However, as will become evident from the following discussions, the distinction between atomic and composite objects as well as the differences resulting from the size of the object space are quite minor compared with the other properties that we use for our classification scheme. Hence, a byproduct of our classification scheme is the unification of concepts and techniques used in data fusion and dependable computation.

Figure 2 shows an example of composite data objects that might be used in voting. The four objects x_1, x_2, x_3, x_4 are infinite sets corresponding to the intervals $[l_1, h_1]$, $[l_2, h_2]$, $[l_3, h_3]$, $[l_4, h_4]$ on the real line. The voting algorithm may depend on the semantics attached to these intervals and on application requirements. For example, if the intervals are interpreted as different views of the safe operating range for a physical parameter, then the interval $[l_2, h_4]$ may be taken as the voting outcome in view of its unanimous designation as being safe. If one of the evaluators (combination of sensors and decision logic) fails so that its corresponding interval is 'way off', then majority consensus can still be reached.

Consider, as another example, the five input objects represented as triangles in Fig. 3. Again, different meanings may be attached to these triangles. Each triangle may simply represent a 3-vector with elements in the range [1, 2), providing three parameters of a system or multiple time-domain measurements of the same parameter by a single sensor. Or the triangles may be outputs of intelligent sensors which extract contour information from visual data. Details of voting will be different with each interpretation. For example, with the 3-vector view, there is majority agreement that the smallest element is 1.5 and the largest element is 1.8. With the visual interpretation, on the other hand, the sensors are unanimous about the contour shape being triangular and have majority agreement on the triangle being isosceles; both of these conclusions may be significant in a target tracking and recognition context.

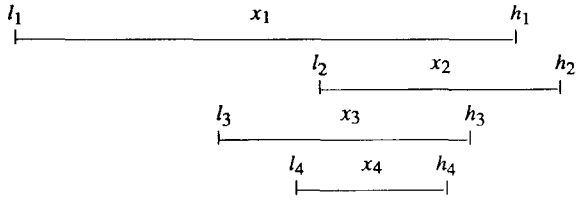

Fig. 2. Voting with composite data objects represented as intervals on the real line.

To accommodate all voting schemes of interest, including those dealing with composite data objects as discussed above, we present the following general definition of weighted voting.

3.1 Definition

Weighted voting—given n input data objects x_1, x_2,...,x_n, and associated non-negative real votes (weights) v_1, v_2,..., v_n, with $\sum_{i=1}^{n} v_i = V$, it is required to compute the output y and its vote w such that y is 'supported by' a number of input data objects with votes totaling w, where w satisfies a condition associated with the desired voting subscheme; e.g., $w > \frac{1}{2}V$ for majority voting, $w > \frac{2}{3}v$ for byzantine voting, $w \geq t$ for t-out-of-V (generalized m-out-of-n) voting, and w corresponding to maximal 'support' among all possible outputs in the case of plurality voting. The term 'supported by' can be defined in several ways, leading to different voting schemes. For example, with exact voting, an input object x_i supports y iff $x_i = y$. With inexact voting, approximate inequality ($\cong$) is defined in some suitable way (e.g., by providing a comparison threshold ε in the case of numerical values or, more generally, a distance measure d in a metric space) and x_i supports y iff $x_i \cong y$. With approval voting, y must be a subset of the approved set of values that x_i defines.

The Byzantine voting scheme in Definition 3.1 is a generalized form of the unweighted version used in distributed computing[17] where n independent n-way voting nodes/sites must arrive at consistent conclusions in the presence of fewer than $n/3$ faulty nodes (Byzantine faulty nodes may try to confuse other nodes by presenting to them inconsistent values). In this generalized version, less than 1/3 of votes/weights can be associated with faulty nodes.

Thus, we view a voter (Fig. 4) as dealing with n input data objects x_i having the associated votes (weights) v_i, $i = 1, 2,..., n$, (i.e., n input data-vote pairs $\langle x_i, v_i \rangle$) and producing the output data-vote pair $\langle y, w \rangle$. The voting algorithm may also produce a set of n 'support bits' s_i, one for each input, that indicate whether a given input 'supports' or 'agrees with' y (as discussed in Definition 3.1. The input votes v_i, $i = 1, 2,..., n$, and the output vote w need not be explicitly represented or even present at all, but rather may be implied.

The reason we have chosen to base our classification scheme on weighted voting is threefold. First, weighted voting is more general than simple voting and thus useful in a wider context. Setting all

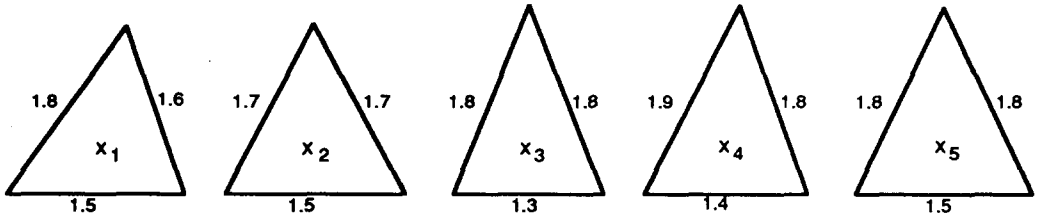

Fig. 3. Example of voting with composite data objects represented as triangles.

the weights equal to 1 yields simple non-weighted voting as a special case. Second, it turns out that in most cases, weighted voting isn't harder to implement than simple voting, especially in software-based implementations (adding an arbitrary weight to a vote tally isn't any harder than adding 1). Third, some constant-weight voting schemes are essentially adaptive over the long run. For example, disabling faulty units or reintroducing repaired units in some n-way voting schemes is equivalent to changing their votes from 1 to 0 or 0 to 1, respectively.

4 THE PROPOSED TAXONOMY

The four main components of a voting algorithm, namely, input data, output data, input votes and output vote, can be used to impose a binary 4-cube classification scheme, leading to 16 classes. Figure 5 depicts the classification scheme.

Briefly, the four dichotomies used in the classification of Fig. 5 are defined as follows. Detailed descriptions and examples can be found in Sections 5, 6, 7 and 8 of the paper.

- *Exact/Inexact*: The exact/inexact dichotomy has to do with whether input objects are viewed as having inflexible values or as representing flexible 'neighborhoods'.
- *Consensus/Mediation*: Consensus voting involves agreement/quorum whereas mediation voting is based on compromise (e.g., choosing the median or mean of real-valued inputs).
- *Oblivious/Adaptive*: The oblivious/adaptive dichotomy corresponds to the v_is being set at design time or allowed to change dynamically (be adjustable or variable).
- *Threshold/Plurality*: Threshold voting requires that w exceed a given threshold whereas plurality voting identifies an output y with highest support from the inputs.

Figure 6 shows the 16 classes resulting from this 4-dimensional classification. Each class corresponds to one of the vertices of a binary 4-cube. The 4-bit vertex labels are interpreted as shown at the bottom left of the diagram. Alternatively, each of the dimensions of this 4-cube may be represented by a letter (E/I for exact/inexact, C/M for consensus/mediation, O/A for oblivious/adaptive, T/P for threshold/plurality), leading to a 4-letter acronym for each of the 16 classes; viz TOCE for 0000, TAME for 0110, POME for 1010, PACE for 1100, etc. Such four-letter acronyms are quite important in the field of computer design and architecture and inventing 16 of them in one go should lead to great endearment in that community!

5 EXACT VS INEXACT VOTING

Depending on how the inputs x_i are viewed, we divide voting algorithms into two classes:

- Exact voting: inputs are viewed as 'exact' or inflexible; y must be equal to some x_i.
- Inexact voting: input objects represent 'neighborhoods' and are considered flexible.

For example, bit-voting algorithms are exact since

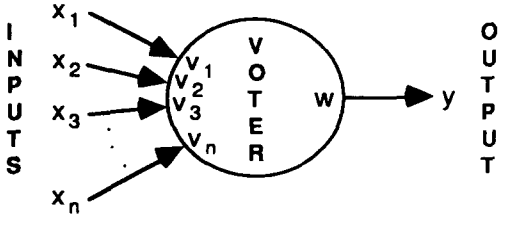

Fig. 4. Elements and parameters of a generalized voter.

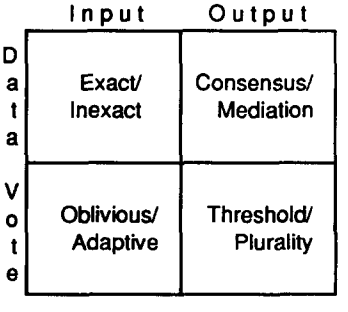

Fig. 5. Classification of voting schemes based on variations in input/output data and input/output votes.

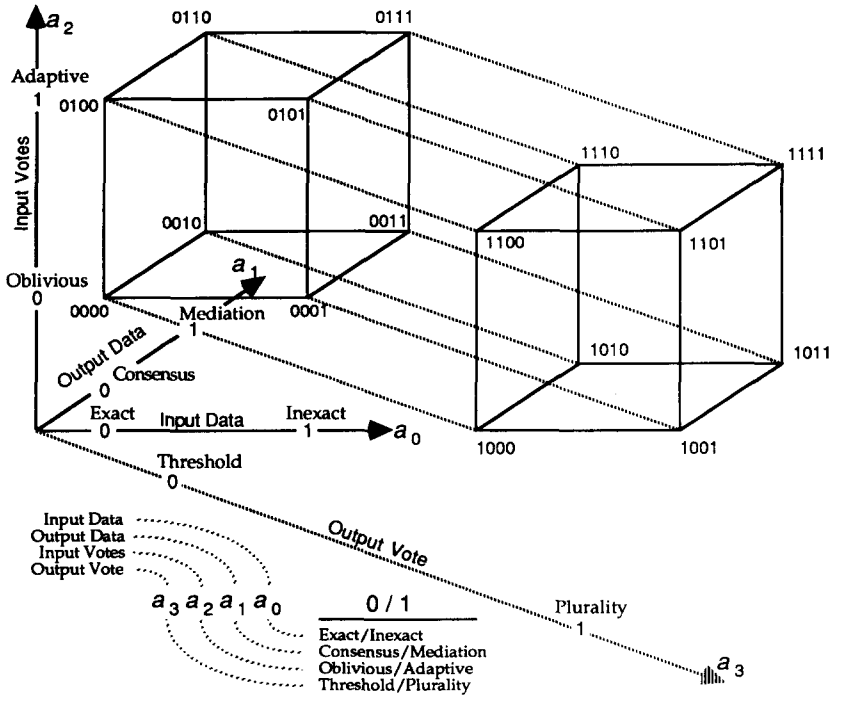

Fig. 6. Binary 4-cube representation of the classification scheme.

the output y must equal one of the inputs x_i (regardless of the voting scheme used, it does not make sense to have $y = 0$ when all the x_i are 1s or vice versa). Word-voting algorithms on floating-point data, on the other hand, may be inexact, since floating-point numbers are expected to contain errors resulting from finite representation and approximate computation. Bounds on such errors can define the neighborhood, or interval, containing the actual intended result.

Exactness or flexibility refers to how the data is viewed for the purpose of voting. This view need not be consistent with other views of the same data. Thus both exact and inexact voting may be meaningful on either a discrete or continuous space, depending on possible output distributions and the utility value assigned to the correct output(s) and various classes of incorrect outputs. For example, integer-valued variables are considered exact in most contexts. However, if integers represent pixel intensities in a gray-scale image, it is perfectly acceptable for a median voting scheme (used to smooth the image by replacing each pixel value with the median of the values in its 8 immediate neighbors or 24 two-step neighbors) to deliver a voting output that is not equal to any of the inputs.

In the simplest case, exact voting involves determining which input value is repeated a given number of times (threshold voting) or more than any other value (plurality voting). This is what has been studied in the past and implemented efficiently in actual systems. However, a more general view does not necessarily lead to inefficiency since the space of possible outputs is limited to no more than n distinct values (the inputs). In this general case, voting can be done in $O(n^2)$ time by computing a 'support' level for each pair of inputs and then choosing an input with maximal total support. More efficient algorithms, e.g., with $O(n \log n)$ or $O(n)$ complexity, may be applicable in special cases such as when the input objects can be sorted or for some threshold voting schemes.

Algorithms for inexact voting are in general more complex than their exact-voting counterparts, primarily due to the non-transitivity of approximate equality. Although the notion of approximate or inexact voting and its applications have been discussed in the literature, the first provably correct inexact voting algorithm was published only recently.[49,52] The algorithm starts by identifying all compatible pairs of inputs, as defined by a generalized distance function and a comparison threshold. It then proceeds to build compatibility classes and a maximal-compatible cover in the same way as is done for minimizing incompletely specified Boolean functions.[30] In the example of Fig. 7, input objects are points on the 2-D plane and the distance function is the Euclidean distance. Each circle encloses a set of points whose pairwise distance is no more than ε. There are 4 maximal-compatible classes. The final voting decision is often based on the inputs in a largest maximal-compatible class (the circle enclosing 5 points in Fig. 7). Details of selection are application-dependent.

The following simple example illustrates the complete procedure.

5.1 Example

Consider the following 5 object-vote pairs (x_i a real value and v_i an integer) as inputs to an inexact 8-out-of-15 (majority) voting algorithm with the comparison threshold of $\varepsilon = 0.02$. The distance function is defined as $d(x_i, x_j) = |x_i - x_j|$.

Objects	Votes
$x_1 = 1.300$	$v_1 = 1$
$x_2 = 1.310$	$v_2 = 2$
$x_3 = 1.330$	$v_3 = 5$
$x_4 = 1.340$	$v_4 = 4$
$x_5 = 1.350$	$v_5 = 3$

The compatible pairs are: (x_1, x_2), (x_2, x_3), (x_3, x_4), (x_3, x_5), (x_4, x_5). The maximal-compatible cover is thus $(x_1, x_2)(x_2, x_3)(x_3, x_4, x_5)$ with class vote tallies being 3, 7 and 12, respectively. Thus, $w = 12$ satisfies the threshold. The maximal-vote selection rule in (x_3, x_4, x_5) yields $y = 1.330$, while the weighted median and weighted mean rules result in $y = 1.340$ and $y = 1.338$, respectively. In this case, the total order imposed on the inputs (sorting) would have yielded a faster algorithm, so the above was used only for illustrating the general method.

It is interesting to note that inexact voting problems with real-valued inputs can be formulated as approval voting problems[49,52] with input sets given by the intervals $[x_i - \varepsilon/2, x_i + \varepsilon/2]$. Figure 8 depicts the 5 input intervals corresponding to Example 5.1 and shows the approval level for each segment and for the three maximal-compatible classes (arrows).

6 CONSENSUS VS MEDIATION VOTING

Depending on how the output data is obtained, there are two classes of voting algorithms:

- Consensus voting: A subset of inputs with votes totaling w 'agrees with' or 'supports' y.
- Mediation voting: The output y minimizes/maximizes an objective function of all inputs.

Although it is possible to consider more complicated combining functions for the votes in consensus voting, we only consider the summing function here. This is consistent with the notion of voting in other contexts. In consensus voting, one must determine the level of support provided by each input to each possible output. When the output space is large, the structure of the space and application semantics should be used to make the computation tractable. For example, we saw at the end of Section 5 that for inexact voting with real-valued inputs, the set of all values supported by each input can be characterized as an interval. Then, one can examine a small number of intervals and interval end points, instead of an infinite number of real values, to determine the output.

Mediation voting covers rank-based voting schemes such as median voting whereby the median of n input values is selected not because of 'majority' or 'maximum' support but as a form of moderation. The objective function minimized in the case of median is the maximum distance between any input and the output. Note that median can be defined for any metric space and not just for numerical inputs (see below). As another example, mean voting minimizes the sum of distances between all inputs and the output. Other objective functions, such as sum of the squares of distances, may be appropriate in other contexts.

Assuming a metric space (i.e., where a distance function is defined on pairs of input objects), we define the generalized median of a set recursively as follows. For a single-element set, that one element is the median. For a 2-element set, the median is defined as a special case depending on the application context. For example, when the objects are points on a Euclidean plane, the median may be defined as the midpoint of the straight line segment connecting the two points. For a set with $n > 2$ elements, the median of the set is defined as the median of the $(n-2)$-element set obtained by removing a pair of points having the largest distance between them from the set (akin to removing the highest and lowest scores in gymnastics competitions). Note that in case of a tie in distances, the particular pair chosen for removal may affect the final result only if the set has been already reduced to 3 or 4 elements.

6.1 Example 1

Consider the 9 input objects (points) shown in Fig. 7. Assume equal votes for all inputs. For this example,

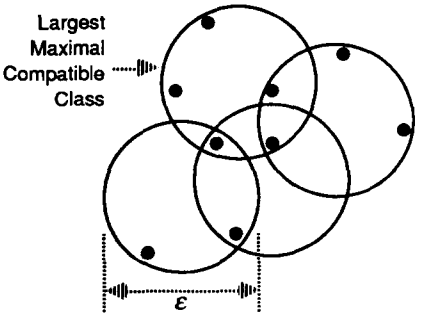

Fig. 7. Maximal-compatible classes for inexact voting on 2-D data points.

the generalized median-finding algorithm described above, successively removes the 4 pairs of points marked with 1, 2, 3, and 4 in Fig. 9, leading to the designation of the remaining point as the 'median'.

Similarly, generalized mean can be defined as an object that has the smallest total distance to all the input objects. In the case of 2-D or 3-D Euclidean space, the generalized mean is the 'center of gravity' for the set of input points, each with its associated 'weight'.

Some consensus voting schemes may be viewed as special cases of mediation voting in the same way that certain exact voting schemes constitute limiting cases of inexact voting (e.g., with $\varepsilon = 0$). The objective function to be maximized for such consensus voting scheme is defined based on 'support'. However, there are consensus voting schemes that cannot be characterized as mediation voting. For example in the approval voting problem depicted in Fig. 8, maximizing the objective function 'total support' (where an output value is supported by an input interval if it is belongs to the interval) leads to the unique solution 1.34 (total support = 12). However, the original formulation of the problem in Definition 5.1 required only majority support which is satisfied by the interval [1.33, 1.34].

Besides pure consensus or mediation voting, it is also possible to have a two-stage voting process whereby first a subset of mutually supportive inputs is identified through consensus voting and then mediation voting is applied to this subset. The reverse process of applying consensus voting to multiple results obtained via mediation voting, though theoretically possible, does not appear to be practically useful.

6.2 Example 2

Again consider the 9 input objects (points) shown in Fig. 7 and assume equal votes for all inputs. Here, one may first form the maximal-compatible sets, as discussed in Section 5, and then find the median in a largest set, as shown in Fig. 10. This method has been found particularly useful for removing inputs that are 'way off' from further consideration (e.g., in some algorithms for synchronizing multiple clocks).

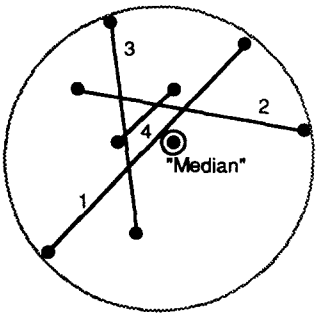

Fig. 9. Generalized median voting on a set of points.

7 OBLIVIOUS VS ADAPTIVE VOTING

Depending on how the input votes v_i are specified or presented, we divide voting algorithms into two classes:

- oblivious voting: the input votes v_i are fixed (i.e., built into the voting algorithm),
- adaptive voting: the votes v_i are stored in writable memory or provided as inputs.

An important special case of oblivious voting is when the input votes v_i are all equal and can thus be ignored. Adaptive voting may be implemented by 'adjustable' votes (stored in writable memory) or by 'variable' votes (presented as inputs along with the data). This distinction is important from the standpoint of voter complexity/cost and algorithm speed. Whereas the voting algorithm itself is the same for adjustable and variable votes, a hardware realization with variable votes needs more I/O pins and thus implies higher cost. Also, with adjustable votes, it may be possible to precompute some intermediate results needed to reach a voting decision

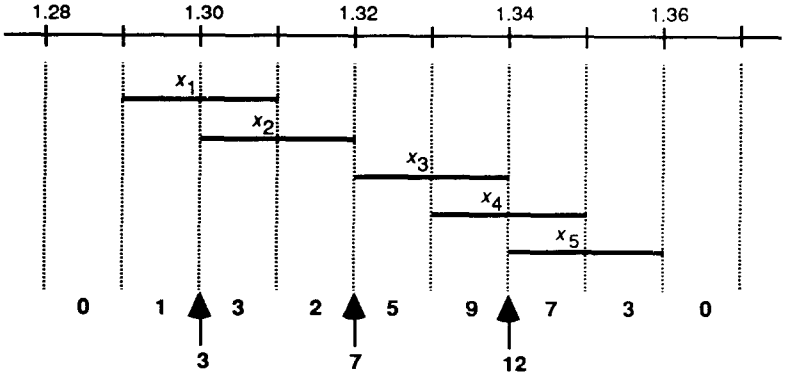

Fig. 8. Approval-voting formulation of an inexact voting problem.

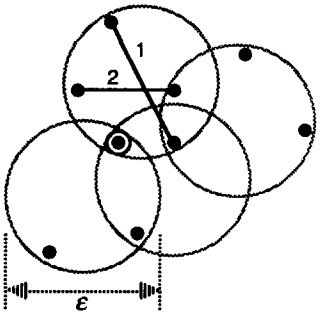

Fig. 10. Generalized median voting on a selected subset of inputs.

at the time when new votes are loaded in order to speed up the subsequent voting process.

A commonly used special case of adaptive voting is when $v_i \in \{0, 1\}$, where $v_i = 0$ corresponds to an input data source that abstains from participation in view of internal conditions (e.g., self-test results) or one that has been disabled by a system-level control function (perhaps due to repeated disagreements with the voting result). An example of this type of adaptive voting is depicted in Fig. 11. In this scheme, known as hybrid redundancy, there are a total of $n = a + s$ modules connected to a voter with adjustable input votes in $\{0, 1\}$. The a active modules have vote weights of 1 and thus influence the voting decision equally. The s spare modules have vote weights of 0, so they have no effect on the voter's output. These spare modules may be completely shut down (cold standby), up and running like the active modules (hot standby) so that they can be switched in with minimal latency when needed, or operating in an intermediate mode (warm standby). The voter itself is really a combination of disagreement detection, switching, and decision elements.

Adaptive voting with completely variable input votes has been used to a lesser extent in practice. A natural environment for the application of such voting schemes is in the author's data-driven dependability assurance method,[43,44] where data objects carry with them 'dependability tags' that convey information about their trustworthiness. As data objects are manipulated, they become less and less trustworthy in view of the potential for error in each such 'dependability-lowering' operation. Dependability tags may be viewed as vote weights, or can be used to derive vote weights, for the data objects when they are used in 'dependability raising' operations exemplified by voting on multiple independent results.

As a more concrete example of this approach to adaptive voting, consider a hybrid software redundancy scheme[54] based on n-version programming and acceptance testing, as depicted in Fig. 12. In n-version programming, several program modules are executed independently and the final result is obtained by voting on the module outputs. In acceptance testing, results obtained from a program module are subjected to an acceptance test and alternate program modules are invoked if the results do not pass the test. Invocation of alternates continues until one produces results that pass the test. Figure 12 shows a particular combination of these methods, where one of the n program modules of a pure n-version scheme is replaced with an acceptance test. Furthermore, the acceptance test (T) is applied to the result obtained by voting on k of the $n - 1$ module outputs (V_1), with the tested result, along with outputs from the remaining $n - 1 - k$ modules entering into a second-level voter (V_2). Assuming program modules M_i with identical reliabilities, an optimal second-level voter in this scheme would attach a vote to its leftmost input that is a function of k, the voting decision in V_1 (e.g., the

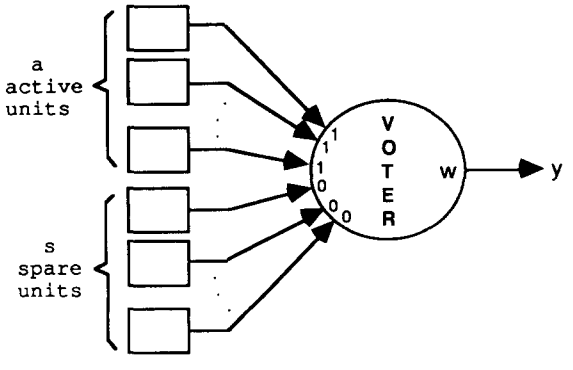

Fig. 11. Hybrid redundancy viewed as an adaptive voting scheme.

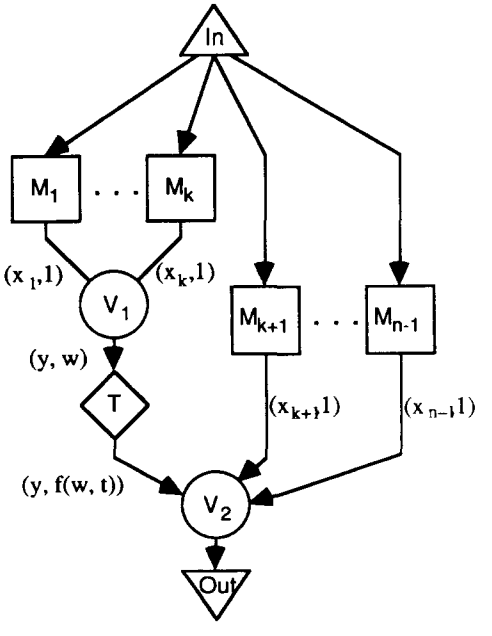

Fig. 12. Adaptive voting is desirable with asymmetric multi-channel computation.

number of disagreements), and the outcome t of the test T. For example, $w+1$ and $w-1$ have been suggested as suitable weights for 'pass' and 'fail' test outcomes, respectively, where w is the weight attached by V_1 to its output. Details are beyond the scope of this paper.

8 THRESHOLD VS PLURALITY VOTING

Finally, depending on how the output vote is handled, two classes of voting algorithms can be distinguished:

- threshold voting: the output vote exceeds a given (implicit or explicit) threshold,
- plurality voting: the sum of votes w is no less than that of any other possible output value.

Common majority voters fall within the threshold voting category, as do m-out-of-n voters, where the data output y implies that at least m of the n data inputs x_i were supportive of y. In both of the above special cases, the output vote is implicit and at most a binary indication of 'quorum' or 'lack of quorum' is provided. With plurality voting, output votes are computed for each candidate output and one of the candidates that obtains the highest vote w is selected. The output vote w may itself be implicit or explicit (presented at the output). Note that the output may be non-unique for both threshold and plurality voting. For example, 2-out-of-5 voting may yield two valid outputs and unweighted 5-way plurality voting may lead to 2 possible outputs with 2 votes each. In such cases, the choice of output may be arbitrary or based on secondary criteria, depending on the application context.

Although many threshold voting schemes of practical interest can be implemented through plurality voting followed by a simple comparison of the resulting output vote with the threshold, the result may be much less efficient that direct implementation of threshold voting (see below). As an example showing that, even ignoring the comparison needed at the end, threshold voting cannot be viewed as a special case of plurality voting, consider a 5-way majority voting scheme. Applying plurality voting to the inputs may result in identifying the sole candidate output that is supported by, say, 4 of the inputs. However, another candidate output supported by 3 of the inputs may be an equally valid result for the majority voter (recall that one input may support several candidate outputs).

Threshold voting is fundamentally simpler than plurality voting.[51] Algorithm 8.1 shows that n-way weighted t-out-of-V exact threshold voting (where 'support' is defined as object equality and such equality can be established in unit time) has time complexity $O(np)$ and needs working storage space for only p objects, where $p = \lfloor V/t \rfloor$. Thus, unless t is much smaller than V, threshold voting is considerably more efficient. As a corollary, weighted majority voting can be performed in linear time with working storage for a single input object. This should be contrasted to the $O(n^2)$ time and $O(n)$ space needed for plurality voting under the same conditions when the object space is not totally ordered.[51] If the object space is ordered, sorting can be used to reduce the time complexity of plurality voting to $O(n \log n)$, which is still significantly higher than $O(np)$ in most cases of practical interest.

8.1 Algorithm

Exact Threshold Voting—Large Unordered Object Space

We need working storage space or 'slots' for $p = \lfloor V/t \rfloor$ different objects $z_1, z_2, \ldots, z_p$, each with an associated vote tally u_j.

1. $z_1 := x_1$; $u_1 := v_1$; $u_j := 0$ $(2 \le j \le p)$
2. for $i = 2$ to n do
3. if $\exists u_j \neq 0$ such that $x_i = z_j$
4. then $u_j := u_j + v_i$
5. else if $\exists u_j = 0$
6. then $z_j = x_i$; $u_j := v_i$
7. else let $m = u_k$ be a minimum of all u_js $(1 \le j \le p)$
8. if $v_i \le m$
9. then $u_j := u_j - v_i$ $(1 \le j \le p)$
10. else $z_k := x_i$; $u_k := v_i$; $u_j := u_j - m$ $(1 \le j \le p)$
11. endif
12. endif
13. endif
14. endfor
15. $u_j := 0$ $(1 \le j \le p)$
16. for $i = 1$ to n do
17. if $\exists j$ such that $x_i = z_j$ then $u_j := u_j + v_i$ endif
18. endfor
19. Output z_j with $u_j \ge t$ ■

Following is a textual description of the algorithm. The first input object and its associated vote are stored in the first slot and all other u_js are initialized to 0, thus designating the remaining $p-1$ slots as empty (Line 1). Input objects $x_2, \ldots, x_n$ are then examined in turn (Line 2). The next input object x_i to be considered is compared to the stored objects (Line 3). If $x_i = z_j$ for some j, then u_j is incremented by v_i (Line 4). If x_i is not equal to any z_j and fewer than p objects have been stored in the p slots, then (x_i, v_i) is stored in an available slot (Line 6). If all of the p slots are occupied, then the minimum vote tally $m = u_k$ for the stored objects is found (Line 7). If $v_i \le m$, then the new input x_i is discarded and all stored vote tallies are reduced by v_i (Line 9). If $v_i > m$, then all vote tallies

are decremented by m and $(x_i, v_i - m)$ replaces one of the objects which is left with 0 vote tally (Line 10). A second pass through the input, comparing each x_i to all remaining z_js, and tallying the actual vote for each z_j, completes the algorithm (Lines 15–18).

8.2 Example

Consider 6-way, 8-out-of-15 voting with the vote weights 4, 3, 3, 2, 2, 1. Take an instance of the voting problem with inputs $(a, 3)$, $(b, 2)$, $(b, 2)$, $(a, 1)$, $(c, 3)$, $(a, 4)$ in presentation (input) order. Since $V = 15$, $t = 8$, and $p = \lfloor 15/8 \rfloor = 1$, a single working storage slot (z_1, u_1) is required. This slot will hold the values $(a, 3)$, $(a, 1)$, $(b, 1)$, (-, -), $(c, 3)$, and $(a, 1)$ successively as we proceed through the steps of the algorithm. Therefore, a is a candidate value for the voting result and a second pass through the input will yield its actual vote tally for comparison with 8.■

9 CONCLUSIONS

We have presented a taxonomy of voting algorithms as a unified view of methods used in multi-source data fusion and dependable multi-channel computation. Although the voting algorithms that have been used in practice occupy a small fraction of the space defined by our taxonomy, other portions of this space are being increasingly populated by new voting schemes as well as generalizations of older ones. Thus, the proposed taxonomy covers virtually all voting schemes proposed to date and also points to areas that have not received due attention in the past.

We have taken a very general view of voting. As such, our discussion of voting covers not only many methods used in connection with multi-sensor data fusion and multi-channel computation, with identical or diverse designs for the hardware or software components, but also other domains where combining or merging of data occurs.[53] One example is image processing filters where during each pass, pixel values may be replaced by values determined from voting on a predefined neighborhood of nearby points.[12] Another example is in distributed fault diagnosis where voting might be used to determine the signature of a fault-free processor from the self-diagnosis signatures of participating nodes.[66] This latter example can be viewed as data fusion, though not as multi-sensor data fusion.

Armed with this unified view of voting in diverse application contexts, one can proceed in many different directions. Clearly, existing voting algorithms can be studied, evaluated, and extended in light of the proposed taxonomy. Voting schemes can be combined into a small number of 'generic' algorithms that can yield the specific algorithms of interest, with little or no overhead, by proper setting of their parameters. It is important to note that voting is but one of many operations needed in data fusion and dependable computations. Combined analyses of voting and other key system functions is also facilitated by the proposed taxonomy. Analysis of real-time scheduling in an environment where voting delays and other factors, such as resource exhaustion or imperfect coverage, contribute to system failure is one example.[50]

REFERENCES

1. Agrawal, P., Fault tolerance in multiprocessor systems without dedicated redundancy. *IEEE Trans. Computers*, **37** (1988) 358–362.
2. Ahamad, M., Ammar, M.H. & Cheung, S.Y., Multidimensional voting. *ACM Trans. Computer Systems*, **9** (1991) 399–431.
3. Ammann, P.E. & Knight, J.C., Data diversity: an approach to software fault tolerance. *IEEE Trans. Computers*, **37** (1988) 418–425.
4. Avizienis, A., Gilley, G.C., Mathur, F.P., Rennels, D.A., Rohr, J.A. & Rubin, D.K., The STAR (self-testing-and-repairing) computer: an investigation of the theory and practice of fault-tolerant computer design. *IEEE Trans. Computers*, **20** (1971) 1312–1321.
5. Avizienis, A., The *N*-version approach to fault-tolerant software. *IEEE Trans. Software Engng*, **11** (1985) 1491–1501.
6. Babaoglu, O., On the reliability of consensus-based fault-tolerant distributed computing systems. *ACM Trans. Computer Systems*, **5** (1987) 394–416.
7. Barbara, D. & Garcia-Molina, H., The reliability of voting mechanisms. *IEEE Trans. Computers*, **36** (1987) 1197–1208.
8. Blough, D. M. & Sullivan, G. F., A comparison of voting strategies for fault-tolerant distributed systems. In *Proc. 9th Symp. Reliable Distributed Systems*, October 1990, pp. 136–145.
9. Blough, D.M. & Sullivan, G.F., Voting using predispositions. *IEEE Trans. Reliab.*, **43** (1994) 604–616.
10. Boyer, R. S. & Moore, J. S., MJRTY—a fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, (ed. R. S. Boyer) Kluwer, Dordrecht, The Netherlands, 1991.
11. Brilliant, S.S., Knight, J.C. & Leveson, N.G., The consistent comparison problem in *n*-version software. *Software Engng Notes, ACM SIGSOFT*, **12** (1987) 29–34.
12. Brownrigg, D.R.K., The weighted median filter. *Comm. ACM*, **27** (1984) 807–818.
13. Campbell, D. & McNeill, T., Finding a majority when sorting is not available. *The Computer J.*, **34** (1991) 186.
14. Chen, L. & Avizienis, A., *N*-version programming: a fault tolerance approach to reliability of software operation. In *Proc. Int. Symp. Fault-Tolerant Computing*, Toulouse, France, 21–23 June 1978, pp. 3–9.
15. Chen, Y. & Chen, T., Implementing fault tolerance via modular redundancy with comparison. *IEEE Trans. Reliab.*, **39** (1990) 217–225.
16. DeSousa, P.T. & Mathur, F.P., Sift-out modular redundancy. *IEEE Trans. Computers*, **27** (1978) 624–627.
17. Dolev, D., Lamport, L., Pease, M. & Shostak, R., The Byzantine generals. In *Concurrency Control and*

Reliability in Distributed Systems, (ed. B. K. Bhargava) Van Nostrand Reinhold, NY, 1987, pp. 348–369.
18. Garcia-Molina, H. & Barbara, D., How to assign votes in a distributed system. *J. Ass. for Computing Machinery*, **32** (1985) 841–860.
19. Gersting, J. L., Nist, R. L., Roberts, D. B. & Van Valkenburg, R.L., A comparison of voting algorithms for n-version programming. In *Proc. 24th Hawaii Int. Conf. System Sciences*, Kauai, Hawaii, 8–11 January 1991 pp. 253–262.
20. Gifford, D. K., Weighted voting for replicated data. In *Proc. 7th ACM SIGOPS Symp. Operating System Principles*, Pacific Grove, CA, December 1979, pp. 150–159.
21. Gries, D., A hands-in-the-pocket presentation of a k-majority vote algorithm. In *Formal Development of Programs and Proofs*, (ed. E. W. Dijkstra) Addison-Wesley, Reading, MA, 1990, pp. 43–45.
22. Hopkins, A.L., Smith, T.B. & Lala, J.H., FTMP—a highly reliable fault-tolerant multiprocessor for aircraft. *Proc. IEEE*, **66** (1978) 1221–1239.
23. Iyengar, S., Sitharama, S., Kashyap, R.L. & Madan, R.N., Special section on distributed sensor networks. *IEEE Trans. Systems, Man, and Cybernetics*, **21** (1991) 1027–1031.
24. Iyengar, S.S., Jayasimha, D.N. & Nadig, D., A versatile architecture for the distributed sensor integration problem. *IEEE Trans. Computers*, **43** (1994) 175–185.
25. Jajodia, S. & Mutchler, D., Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. Database Systems*, **15** (1990) 230–280.
26. Johnson, B. W., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, 1989.
27. Kanekawa, N., Maejima, H., Kato, H. & Ihara, H., Dependable onboard computer systems with a new method—stepwise negotiating voting. In *Proc. Int. Symp. Fault-Tolerant Computing*, Chicago, 21–23 June 1989, pp. 13–19.
28. Kieckhafer, R.M. & Azadmanesh, M.H., Reaching approximate agreement with mixed-mode faults. *IEEE Trans. Parallel Distributed Systems*, **5** (1994) 53–63.
29. Knight, J.C. & Leveson, N.G., An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. Software Engng*, **SE-12** (1986) 96–109.
30. Kohavi, Z., *Switching and Finite Automata Theory*, McGraw-Hill, NY, 1978, pp.333–347.
31. Krol, T., Interactive consistency algorithms based on voting and error-correcting codes. In *Proc. 25th Int. Symp. Fault-Tolerant Computing*, Pasadena, CA, 27–30 June 1995, pp. 89–98.
32. Lei, C.-L. & Liaw, H.-T., Efficient parallel algorithms for finding the majority element. *J. Info. Science and Engng*, **9** (1993) 319–334.
33. Leveson, N.G., Cha, S.S., Knight, J.C. & Shimeall, T.J., The use of self checks and voting in software error detection: an empirical study. *IEEE Trans. Software Engng*, **16** (1990) 432–443.
34. Lorczak, P. R., Caglayan, A. K. & Eckhardt, D. E., A theoretical investigation of generalized voters for redundant systems. In *Proc. Int. Symp. Fault-Tolerant Computing*, Chicago, 21–23 June 1989, pp. 444–451.
35. Losq, J., A highly efficient redundancy scheme: self-purging redundancy. *IEEE Trans. Computers*, **25** (1976) 569–578.
36. Luo, R.C. & Kay, M.G., Multisensor integration and fusion in intelligent systems. *IEEE Trans. Systems, Man, and Cybernetics*, **19** (1989) 901–927.
37. Marzullo, K., Tolerating failures of continuous-valued sensors. *ACM Trans. Computer Systems*, **8** (1990) 284–304.
38. Mathur, F. P. & Avizienis, A., Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair. In *AFIPS Conf. Proc.*, vol. 36, AFIPS Press, Montvale, NJ, 1970, pp. 375–383.
39. Mathur, F.P. & DeSousa, P., Reliability modeling and analysis of general modular redundant systems. *IEEE Trans. Reliab.*, **R-24** (1975) 296–299.
40. McAllister, D.F., Sun, C.-E. & Vouk, M.A., Reliability of voting in fault-tolerant software with small output spaces. *IEEE Trans. Reliab.*, **39** (1990) 524–534.
41. Misra, J. & Gries, D., Finding repeated elements. *Science of Computer Programming*, **2** (1982) 143–152.
42. Moser, L. E., Kapur, V. & Melliar-Smith, P. M., Probabilistic language analysis of weighted voting mechanisms. In *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990, pp. 67–73.
43. Parhami, B., A framework for the study of computer system dependability. In *Proc. 23rd Asilomar Conf. Signals, Systems, and Computers*, Pacific Grove, CA, 30 October–1 November 1989, pp. 1017–1021.
44. Parhami, B., A data-driven dependability assurance scheme with applications to data and design diversity. In *Dependable Computing for Critical Applications* (Dependable Computing and Fault-Tolerant Systems, vol 4), Springer-Verlag, Wien, 1991, pp. 257–282.
45. Parhami, B., Voting networks. *IEEE Trans. Reliab.*, **40** (1991) 380–394.
46. Parhami, B., The parallel complexity of weighted voting. In *Proc. Int. Symp. Parallel and Distributed Computing and Systems*, Washington, DC, 8–11 October 1991, pp. 382–385.
47. Parhami, B., High-performance parallel pipelined voting networks. In *Proc. Int. Parallel Processing Symp.*, Anaheim, CA, 30 April–2 May 1991, pp. 491–494.
48. Parhami, B., Design of m-out-of-n bit-voters. In *Proc. 25th Asilomar Conf. Signals, Systems, and Computers*, Pacific Grove, CA, 4–6 November 1991, pp. 1260–1264.
49. Parhami, B., Optimal algorithms for exact, inexact, and approval voting. In *Proc. Int. Symp. Fault-Tolerant Computing*, Boston, USA, 8–10 July 1992, pp. 444–451.
50. Parhami, B. & Hung, C. Y., Scheduling of replicated tasks to meet correctness requirements and deadlines. In *Proc. 26th Hawaii Int. Conf. System Sciences*, vol. 2, Kihei, Hawaii, 5–8 January 1993, pp.506–515.
51. Parhami, B., Threshold voting is fundamentally simpler than plurality voting. *Int. J. Reliab., Quality and Safety Engng*, **1** (1994) 95–102.
52. Parhami, B., Voting algorithms. *IEEE Trans. Reliab.*, **43** (1994) 617–629.
53. Parhami, B., Multi-sensor data fusion and reliable multi-channel computation: Unifying concepts and techniques. In *Proc. 29th Asilomar Conf. Signals, Systems, and Computers*, Pacific Grove, CA, 30 October–1 November 1995.
54. Parhami, B., Design of reliable software via general combination of n-version programming and acceptance testing. submitted for publication.
55. Paris, J.-F., Voting with a variable number of copies. In *Proc. Int. Symp. Fault-Tolerant Computing*, Vienna, 1–3 July 1986, pp. 50–55.
56. Pierce, W. H., Adaptive decision elements to improve

the reliability of redundant systems. *IRE Int. Conv. Record*, March 1962, pp. 124–131.
57. Shin, K.G. & Dolter, J.W., Alternative majority-voting methods for real-time computing. *IEEE Trans. Reliab.*, **38** (1989) 58–64.
58. Siewiorek, D.P. & McCluskey, E.J., Switch complexity in systems with hybrid redundancy. *IEEE Trans. Computers*, **22** (1973) 276–282.
59. Siewiorek, D.P. & McCluskey, E.J., An iterative cell switch design for hybrid redundancy. *IEEE Trans. Computers*, **22** (1973) 290–297.
60. Siewiorek, D.P., Reliability modeling of compensating module failures in majority voted redundancy. *IEEE Trans. Computers*, **24** (1975) 525–533.
61. Siewiorek, D.P., C.vmp: a voted multiprocessor. *Proc. IEEE*, **66** (1978) 1190–1198.
62. Siewiorek, D. P. & Swarz, R. S., *Reliable Computer Systems: Design and Evaluation*, 2nd edition, Digital Press, Bedford, MA, 1992, pp. 138–146.
63. Sklaroff, J.R., Redundancy management techniques for space shuttle computers. *IBM J. Research and Development*, **20** (1976) 20–28.
64. Spasojevic, M. & Berman, P., Voting as the optimal static pessimistic scheme for managing replicated data. *IEEE Trans. Parallel Distributed Systems*, **5** (1994) 64–73.
65. Stotts, L. & Stewart, C., Sensor fusion. In *27th Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, 1–3 November 1993, pp. 258–304.
66. Su, S.Y.H., Cutler, M. & Wang, M., Self-diagnosis of failures in VLSI tree array processors. *IEEE Trans. Computers*, **40** (1991) 1252–1257.
67. Tong, Z. & Kain, R.Y., Vote assignments in weighted voting mechanisms. *IEEE Trans. Computers*, **40** (1991) 664–667.
68. Voges, U., Use of diversity in experimental reactor safety systems. In *Software Diversity in Computerized Control Systems*, Springer-Verlag, Wien, 1988, pp. 29–49.
69. von Neumann, J., Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, (Annals of Mathematics Studies, no 34) (eds C. E. Shannon and J. McCarthy) Princeton Univ. Press, 1956, pp. 43–98.
70. Wensley, J.H., Lamport, L., Goldberg, J., Green, M.W., Levitt, K.N., Melliar-Smith, P.M., Shostak, R.E. & Weinstock, C.B., SIFT: design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE*, **66** (1978) 1240–1255.
71. Wensley, J.H. & Harclerode, C.S., Programmable control of a chemical reactor using a fault tolerant computer. *IEEE Trans. Industrial Electronics*, **29** (1982) 258–264.