Performance Analysis of a Hierarchical Quorum Consensus Algorithm for Replicated Objects

Akhil Kumar
Graduate School of Management
Cornell University, Ithaca, NY 14853.
(Bitnet: akhil@crnlgsm)

Abstract

This paper describes a new class of algorithms for synchronizing a large number of identical copies of an object and evaluates its performance. The standard quorum consensus method becomes an expensive means of maintaining consistency across multiple copies of an object if the number of copies is large because it typically requires at least a majority of sites to be involved in a write operation. Our method is based on organizing a group of objects into a multi-level hierarchy, and extending the quorum consensus algorithm to such an environment. A performance comparison, in terms of availability and message cost, against majority voting and dynamic voting is carried out using a simulation model. While our method uniformly outperforms the other two methods in terms of average message cost, no single method was found to dominate in terms of availability. The conditions under which each method performed well were identified.

1 Introduction

Replication of data increases system reliability in order to satisfy the high up-time requirements in various on-line applications. Clearly, if copies of a file reside on several computers with independent failure modes, then the file system would be more reliable. The disadvantage, however, is that the copies must be kept mutually consistent by synchronizing transactions at different sites so that a global serialization order is ensured. For instance, two independent transactions must not be allowed to simultaneously update different copies of the same file. Hence, the concurrency control algorithm becomes more complex and also more expensive to run. The additional communications and processing cost arises because several rounds of messages must be exchanged with other sites during the execution of the algorithm.

The majority consensus method [THOM79], and the quorum consensus method [GIFF79] are well-known algorithms for replicated data concurrency control. Both involve assigning votes to sites, and defining a read and a write quorum. A read operation must assemble enough votes to form a read quorum, while a write must assemble a write quorum. By defining the read and write quorums appropriately, serial consistency or one-copy serializability [BERN83] is achieved. In the majority consensus method, for instance, each site is assigned a single vote, and a majority of votes must be assembled to perform any operation. Subsequently, researchers have proposed many variants of the basic algorithm such as: dynamic voting [DAVC85, JAJO87, PARI88], voting with witnesses [PARI86], the missing writes algorithm [EAGE83] and the views approach [ELAB85]. Protocols for dynamically changing votes and quorum sizes are discussed in [HERL87, BARB86].

A major problem with the quorum consensus method is that the number of objects required for a quorum increases linearly with the total number of objects. Consequently, to perform a write operation on a group of 100 objects would require a quorum of at least 51 objects. With the proliferation of distributed systems, the need for synchronization among such a large number of objects is likely to grow in various application areas.

Theoretically, one could reduce the message traffic in the quorum consensus method by assigning unequal votes to the various copies of an object as suggested in [KUMA88]. However, this optimization does not scale very well and will not be helpful if the number of objects is large, say 50 or 100, because it will exhibit drawbacks of a centralized system, such as excessive reliance on one or a few sites.

Our objective is to develop a synchronization scheme that scales well even when a large number of objects is present. Moreover, it should result in high availability and minimize the number of messages necessary for synchronization. Consequently, we pose the following question: given \( n \) identical objects, what is the minimum number of objects required in order to synchronize both read and write operations in a fully distributed algorithm, i.e. one
This paper generalizes the quorum consensus scheme into a multi-level algorithm called hierarchical quorum consensus and evaluates its performance against two other methods: majority voting and dynamic voting. To illustrate the basic idea behind our algorithm, consider a group of nine objects organized into three subgroups, each of size three (see Figure 1). Our algorithm allows an object to assemble a quorum consisting of two objects each from any two subgroups. The size of a quorum is thus reduced to four objects, instead of five which would be the case in the majority consensus algorithm. It is easy to see that all quorums produced by this algorithm will intersect each other, and therefore the algorithm is correct. Moreover, an identical collection of intersecting quorums can not be produced by any vote assignment within the framework of the quorum consensus method. On the other hand, our method is unable to form a quorum with all three objects from subgroup 1, and one each from subgroups 2 and 3. In this situation, majority voting is better. This trade-off is analyzed in the paper.

The organization of this paper is as follows. Sections 2 and 3 discuss the basic principle behind the algorithm, along with a proof of correctness, the implementation details, and a brief discussion of its properties. The performance study is presented in Section 4. It consists of the simulation model, the results of our experiments, and an analysis of the results. Section 5 concludes the paper.

2 Basic Concepts

We first describe the quorum consensus algorithm, then discuss the principle behind our algorithm, and show its correctness.

Basic Quorum Consensus algorithm [Giff79]:
This algorithm describes how access to a group of o replicated objects can be synchronized. A read transaction must lock q_r copies, while a write must lock q_w copies, such that:

\[ q_r + q_w > o, \text{ and } 2q_w > o. \]

Basically, a write transaction must assemble a quorum of q_w copies, while a read must gather at least q_r copies, and return the value of the copy with the most recent version number. (A version number is maintained with each copy, and this number is incremented every time an update takes place). Under these conditions, the read and the write quorums are said to intersect, i.e., concurrent read and write access to non-intersecting sets of copies is prevented. Moreover, any set of q_r copies will always contain at least one copy with the most recent version.

Our algorithm is based on logically organizing a set of objects into a multi-level tree (of depth n) with the root as level 0. The physical objects are stored in the leaves of this tree, or at level n. Thus, the root has l_1 subobjects (or logical objects) at level 1, and each level 1 logical object in turn has l_2 subobjects at level 2, and so on. Consequently, there are l_n level n (physical) objects within each level n-1 object. Therefore, the total number of physical objects is l_1l_2...l_n. For example, Figure 1 is a two-level hierarchy where the root has 3 level 1 subobjects, each containing 3 physical objects. To perform read and write operations, appropriate quorums must be assembled by traversing the hierarchy from the root to the level of the physical objects, i.e. the bottom level. We first define the concept of a quorum and then derive the conditions for serial consistency in this context.

Definition 1: A read (write) quorum at level i is assembled by gathering r_i (w_i) subobjects of a level i-1 object.

Notice that this is a recursive definition. Therefore, each level i subobject would in turn assemble r_{i+1} subobjects at level i+1, and so on.

Definition 2: We say that access to a logical object at level i is serially consistent if only one write operation can be performed on the subobjects within that object at a time, and reads and writes intersect.

Theorem 1: A concurrency control scheme which requires each read operation to (recursively) assemble a quorum of r_i objects at level 1, and each write operation to (recursively) assemble a quorum of w_i copies at level 1 is correct if for all levels i, i=1,..., n, the read quorum r_i and the write quorum w_i are such that:

\[ r_i + w_i > l_i, \text{ and } 2w_i > l_i. \]
Proof: (by induction) The above theorem is proved by starting at the lowest level of the tree (level n) and deducing the conditions required for serial consistency at successively higher levels.

(Base step) Consider a read or a write access to a logical object at level n-1. Each such object consists of \( n_i \) physical objects. We first show that within any logical object at level n-1, a read and a write transaction will intersect and two writes will also intersect. This follows directly from the hypothesis: \( r_n + w_n > l_n \). This means that a group of \( n_i \) objects at level n can be accessed by only one write transaction (or one or more read transactions if no writes are in progress) at a time. Consequently, access to the (logical) objects at level n-1 is serially consistent.

(Inductive step) Next, we show that if access to objects at level i is serially consistent, access at level i-1 would also be serially consistent with an appropriate choice of \( r_i \) and \( w_i \). Each level i-1 object consists of \( n_i \) level i subobjects. Since access to level i objects is serially consistent, each level i object has the property that at most one write operation can be performed on it at a time, while a read and a write operation are prevented simultaneously. Now, an access to an object at level i-1 will involve subsequent accesses to its sub-objects at level i. If a read operation at level i-1 assembles \( r_i \) copies of level i subobjects, and a write operation assembles \( w_i \) copies such that \( r_i + w_i > l_i \), then two simultaneous write accesses or a simultaneous read and a write access by different transactions can be prevented. This makes access to level i-1 objects serially consistent.

Therefore, by induction on i, it follows that if a read operation (recursively) assembles \( r_i \) logical objects at level 1, and a write operation assembles \( w_i \) logical objects at level 1, then all access is serially consistent. Hence, the theorem is proved.

Corollary 1: The number of physical objects in a read quorum is: \( r_1 r_2 \ldots r_n \), while the size of a write quorum is \( w_1 w_2 \ldots w_n \), where \( r_i \) (or \( w_i \)) is the size of the read (write) quorum at level i. Also, the write quorum is at least as large as the read quorum.

Proof: (for read quorum) The root must assemble \( r_1 \) level 1 objects. Each selected level 1 object must in turn assemble \( r_2 \) level 2 objects, and so on. Therefore, the actual number of physical objects assembled in a read quorum is \( r_1 r_2 \ldots r_n \). (The proof for the size of the write quorum is identical).

The write quorum is at least as large as the read quorum because each individual \( w_i \) is at least equal to the corresponding \( r_i \).

3 Algorithm

In Section 3.1 we describe the implementation details of our algorithm. Then in Section 3.2, we give an illustrative example, and finally, two properties of this algorithm are discussed in Section 3.3.

3.1 Hierarchical Quorum Consensus (HQC)

As mentioned above, the physical objects are organized into an n-level hierarchy (with the root as level 0) such that each level i-1 logical object contains \( n_i \) subobjects. Physical objects are present only in the leaf level (level n) of this hierarchy.

Each physical object is assigned an n-subscript label (or identifier) of the form \( a_{1i}, a_{12}, \ldots, a_{1m} \), where each \( a_{ij} \) varies between 1 and \( n_j \) for \( j \) varying from 1 to n. For instance, in a two-level hierarchy of 9 objects such as Figure 1, with \( l_1 = 3 \) and \( l_2 = 3 \), the objects are labelled as \( a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33} \). Similarly, in a 3-level hierarchy of 27 objects, the objects are labelled as \( a_{1i}, a_{2j}, a_{3k} \), where \( i, j, \) and \( k \) vary from 1 to 3.

The basic idea behind the algorithm is that a read action must (recursively) assemble a quorum of \( r_1 \) level 1 objects (i.e., each selected level 1 object must assemble \( r_2 \) level 2 objects, and so on). Similarly, a write action must (recursively) assemble a quorum of \( w_1 \) level 1 objects. For example, in the 2-level hierarchy above, assume that \( r_1 = 2, r_2 = 1, w_1 = 2 \) and \( w_2 = 1 \) are each set to 2. Clearly, 4 objects are required for read or write synchronization, and one possible quorum is: \( a_{11}, a_{12}, a_{21}, a_{22}, a_{31}, a_{32} \). For another example, consider the 3-level hierarchy mentioned above, and assume that each of \( r_1, r_2, r_3, w_1, w_2 \) and \( w_3 = 2 \). In this case 8 objects are required for synchronization and a possible quorum is: \( a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33} \).

The complete listing of the algorithm is given in pseudo-code below and a brief description follows.

Algorithm 1

```
assemble(1, q[1]);
  if (num_obj[1] ≥ q[1])
    return(yes) /* successful */
  else
    return(no) /* unsuccessful */
assemble(level_num, q)
  i[1:level_num] = 1;
  num_obj[1:level_num] = 0;
  push(stack, "marker", level_num);
  while (num_obj[1:level_num] < q) and
    i[1:level_num] ≤ i[level_num]]
  if (level_num < n)
    assemble(level_num+1, q[level_num+1])
  else
    lock object o[i[1],i[2],...,i[n]];
    push(stack, "o[i[1],i[2],...,i[n]]");
    num_obj[1:level_num]++;
  i[1:level_num]++;
}
/* end of while loop */
if (num_obj[1:level_num] < q)
  release_locks("marker", level_num);
else
```

380
increment num_obj[level_num - 1];

release_locks(marker)
X = pop(stack);
while (X ≠ marker) {
    if (X = object name) /*if 'pop'ed item is* /
        unlock X; /*an object, unlock it */
    X = pop(stack);
}

The variable q[k] is the size of a quorum at level k, and num_obj[k] is the number of objects assembled at level k each time procedure assemble is called. Finally, i[k] represents the total number of subobjects within a level k-1 object, and j[k] is a counter which keeps track of the number of them examined. The algorithm makes a call to the recursive procedure assemble to form a quorum of size q[1] at level 1, and then checks whether it has been successful. Procedure assemble calls itself recursively with two parameters: level number and the required size of the quorum at that level. In case it is not possible to assemble the appropriate quorum at any level in the hierarchy, all locks obtained within that level are released. This is accomplished by maintaining a stack, and "pushing" a marker entry with the corresponding level number into it. The release_locks procedure is invoked to "pop" entries from the stack until the marker entry, and to unlock all objects removed from the stack in the process.

3.2 Example

A collection of 27 objects can be organized into a three-level hierarchy such that l_1, l_2, and l_3 are each 3. Table 1 shows the various possible combinations of read and write quorum values at each level, and also the total number of objects required for read and write synchronization. Notice that our method allows a write quorum to be formed with as few as 8 objects, while the majority voting method would require 14.

3.3 Properties

This algorithm has the following properties shown in [KUMA89]. First, the hierarchical consensus algorithm does not have an equivalent single-level representation. Second, the minimum number of messages required for synchronization among objects is given by Ω(Ω). An algorithm for determining the correct number of levels to minimize the number of messages is also given in [KUMA89].

<table>
<thead>
<tr>
<th>No.</th>
<th>r_1</th>
<th>w_1</th>
<th>r_2</th>
<th>w_2</th>
<th>r_3</th>
<th>w_3</th>
<th>r</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Possible Read and Write Quorums in a 3-level hierarchy of 27 objects

4 Performance Study

4.1 Introduction

This section is devoted to our performance study. The study compares the hierarchical quorum consensus algorithm against two other quorum-based methods: majority voting and dynamic voting. The majority voting algorithm is identical to quorum consensus, described in Section 2.1, with all sites being assigned equal votes, and both read and write quorums made equal to a majority of votes. The dynamic voting algorithm, described in [JAJO87], requires sites to assemble quorums before performing any operation. However, the quorums are assembled in a different manner. Each site must keep the following additional information: a version number of the object and the number of sites that were involved in the most recent quorum that was formed at that site for the object. The version number is incremented every time a new value is written to an object. To form a quorum, a site must examine the version numbers of all the sites that it can reach, select the latest version number and identify the number of sites that have the latest version number. If at least a majority of the sites involved in the most recent quorum can be reached, then a new quorum can be formed; otherwise, it is disallowed.

The three algorithms have been compared, by a simulation, on the following two criteria: average number of messages and availability. The average number of messages is the average number of additional sites that must be involved in forming a quorum. Availability is the long-term fraction of time (or probability) for which at least one site is able to form a quorum.

We considered two 9-site networks with different degrees of connectivity: low connectivity and high connectivity. The average number of direct links per node is used as a surrogate for connectivity. To compute connectivity, the number of direct links for each node is first determined by examining the network. Then, this number is aggregated across all nodes and the sum divided by the total number of nodes to obtain the average. In the low connectivity network this figure is 2.67, while in the high connectivity case it is 4.0.

The performance model is described in Section 4.2,
the results of the experiments in Section 4.3, and a discussion follows in Section 4.4.

4.2 Performance Model

The simulation model considers both link and site failures. The reliability of both links and sites are defined in terms of two parameters: mean time between failures (MTBF) and mean time to repair (MTTR). MTBF is the average length of time a link (or site) is up before the next failure occurs, while MTTR is the average time taken to repair it. We assume that the length of the up and down intervals follows an exponential distribution. The reliability $R$, of a link (or site) is the probability that it is up at any given instant, and is computed from MTBF and MTTR as:

$$R = \frac{MTBF}{MTBF + MTTR}$$

Therefore, if MTBF is 99, and MTTR is 1, it corresponds to a reliability of 0.99.

Our simulator is event-driven, and the inputs to it are: the network description (i.e. sites and links), MTBF and MTTR values for sites and links, and duration of the simulation. The outputs of the simulation are the average availability and message cost for each method. Initially, it is assumed that all components (sites and links) are up, and failure times for each component are randomly generated from exponential distributions with the appropriate MTBF values. A “reach” matrix is also generated indicating the set of sites that can be reached from each site (either directly, or via another node).

The clock is advanced by one time instance, and all events that are to occur at that time are handled. There are four possible events: an “up” site may go down, a “down” site may come up, an “up” link may go down, or a “down” link may come up. Handling an event consists of noting the new state of the network as a result of the event, generating a new event for that component, and producing a revised reach matrix. Finally, at each clock instance, each site runs a transaction and determines whether a quorum can be formed. At least one site must be able to form a quorum for the system to be available. The number of additional sites needed to form a quorum is also determined. The simulator keeps statistics on availability and messages.

The simulation is repeated for each algorithm, and terminates when the upper limit for the length of the simulation is reached. At this point, the average availability and the average number of messages are computed for each algorithm.

4.3 Results

Two separate sets of experiments were performed for the low and high connectivity networks. For each network, appropriate MTBF and MTTR pairs of values were selected to correspond to three values of link reliability: 0.99, 0.90, 0.80. In any given experiment, all links were assumed to have identical reliability. Similarly, all sites were also treated as identical in an experiment. For each link reliability value, the site reliability was varied over a set of values ranging from 0.80 to 0.99, and the availability and message cost of the three methods was computed by the simulator. The results are discussed below, separately for the low and high connectivity networks.

4.3.1 Low Connectivity Network

In this section we discuss results of experiments on a 9-site network with a connectivity per node of 2.67. The availability percentage was plotted against individual site reliability for the three methods. (An availability percentage of 90 means that the system is available 90% of the time). The plots in Figures 2, 3, and 4 correspond to link reliabilities of 0.99, 0.90 and 0.80, respectively. Figure 5 is a plot for the average message cost versus site reliability for the three methods. The message cost was averaged across the three link reliability values because it stays constant in the majority voting and hierarchical quorum consensus (HQC) algorithms, and varies very slightly for different values of link reliability in the dynamic voting algorithm.

Figure 2 shows that when link reliability is low (0.80), the HQC method dominates the other two methods for the entire range of site reliability values. On the other hand, when link reliability is very high (0.99), as in Figure 4, dynamic voting is superior to the other two methods. Figure 3 shows that when the link reliability is 0.9, the dynamic voting method is better for site reliabilities in the range of 0.8 to 0.9, but HQC gives a higher availability when site reliability is greater than 0.9.

These results show that in a low connectivity network, the relative performance of the HQC method is sensitive to both link and site reliability. The trade-offs between HQC and dynamic voting are as follows. The HQC method requires that at least 4 sites be up and connected with each other as a prerequisite for a quorum to be formed. In a poorly connected network with relatively low link reliability, partitions will occur frequently, and often the largest partition will contain fewer than 4 “up” sites. A high link reliability increases the probability that if 4 sites are up they will be able to communicate with one another. On the other hand, the dynamic voting algorithm modifies itself according to the state of the network and can even form a quorum in a partition containing only two “up” sites.

The second trade-off pertains to how rapidly the state of the network changes between two successive transactions. Dynamic voting requires that at least a majority of the sites that were involved in the most recent quorum be available for the next quo-
Figure 2: Low Connectivity Network
(link reliability = 0.99)

Figure 3: Low Connectivity Network
(link reliability = 0.90)

Figure 4: Low Connectivity Network
(link reliability = 0.80)

Figure 5: Low Connectivity Network

Figure 6: High Connectivity Network
(link reliability = 0.99)

Figure 7: High Connectivity Network
(link reliability = 0.90)

Figure 8: High Connectivity Network
(link reliability = 0.80)

Figure 9: High Connectivity Network

Legend:
- ▲ Hierarchical QC
- ● Dynamic Voting
- ■ Majority Voting
rurn. Therefore, it is evident that a sudden change in the state of the network from, say, 9 "up" and connected sites, to 4 "up" sites in the largest partition will prevent a quorum from being formed. On the other hand, HQC may continue to work in spite of such a change. Our results show that in a weakly connected network, HQC wins the trade-offs if the link reliability is very high (0.99), and also when both link and site reliabilities are greater than 0.9.

It should also be noted that the availability of majority voting is the least for all values of link and site reliability. This is to be expected because majority voting requires 5 sites to be working and in the same partition. Since the connectivity of the network is low, this algorithm performs badly, although the gap between majority voting and the other two algorithms shrinks as the site reliability increases.

Finally, the message cost comparison in Figure 5 shows that HQC dominates the other two methods in terms of message cost, and that dynamic voting is uniformly the most expensive. The message cost of HQC and majority voting is fixed, while for dynamic voting it varies as a function of site reliability in an almost linear manner.

4.3.2 High Connectivity Network

The experiments described above were repeated for a 9-site network in which the average connectivity per node was 4.0.

Again, availability of the three methods was simulated for various combinations of link and site reliabilities. The availability is plotted against site reliability for three different values of link reliability in Figures 6, 7, and 8. The average message cost is plotted in Figure 9. In all three availability plots the dynamic voting algorithm performs the best, while HQC performs the worst.

Dynamic voting modifies itself to changes in the state of the network and can continue to work even with as few as two sites. It is uniformly superior to HQC because, in a high connectivity network, it always wins the trade-offs discussed above. The main reason for this is that in such a network partitions are much less frequent than in a low connectivity network, and therefore, rapid changes in the state of the network are rare. Hence, dynamic voting outperforms the other two algorithms by a wide margin.

Now we turn to examine why majority voting is uniformly better than HQC. In a well-connected network, partitions occur less frequently than in a weakly connected network because each site has links with several other sites. This is especially true if the links are reliable. Moreover, even if partitions do occur, chance are high that at least one partition would have 5 or more sites. The main trade-off between majority voting and HQC is as follows. Majority voting can form a quorum if there is any partition with at least 5 working sites, while HQC can form a quorum even with 4 sites under certain conditions and might not do so with 5 sites under other conditions. When connectivity is high, and the links are highly reliable, majority voting wins this trade-off by a large margin as shown in Figure 6. On the other hand, when link reliability drops, the advantage of majority voting diminishes as illustrated in Figures 7 and 8.

The plot of message cost versus site reliability (Figure 9) is similar to the one for the low connectivity network. Again, HQC is the least expensive, and dynamic voting considerably more expensive.

4.4 Discussion

The following conclusions can be drawn from the experiments. First, it is evident that HQC is the cheapest method in terms of message cost. The savings are substantial in comparison with dynamic voting, and will increase further if the number of objects is larger. Secondly, HQC also gives the best availability, among the three algorithms studied, for a low connectivity network with highly reliable links. On the other hand, dynamic voting produces the highest availability when either the network connectivity is high, or the links are less reliable in a low connectivity network. Thirdly, HQC uniformly outperforms majority voting in a low connectivity network by a wide margin, but the opposite is the case in a high connectivity network, although the gap is much smaller. In summary, no single method was uniformly superior, and the conditions under which each performed well were identified.

Finally, another factor that affects the performance of dynamic voting is the frequency at which transactions are performed on the replicated object. If the frequency is small, then the amount of change in the network state between two consecutive transactions would be larger than if the frequency was high. Since the algorithm performs better when changes in network state are gradual, it is clear that a high frequency of transactions would lead to better performance. In our experiments, transactions were performed at every clock instance, and MTBF and MTTR values were kept relatively large to reduce the number of events at each instance. Therefore, the results for dynamic voting are on the optimistic side. Studying how performance changes with the frequency of transactions is left as a future exercise. We anticipate that dynamic voting would become less attractive if the frequency of transactions were low. This factor does not affect the other two algorithms because they are not dynamic.

5 Conclusion

In a replicated data environment, multiple copies of an object must be kept synchronized by means of
a synchronization algorithm. The quorum consensus algorithm incurs a high cost of messages if the number of objects to be synchronized is large. In this paper we introduced a new class of algorithms, also based on voting, that is more general than the standard quorum consensus method because it allows a larger collection of intersecting quorums to be formed.

A performance study comparing the hierarchical quorum consensus (HQC) algorithm against two other quorum-based methods was carried out for two 9-site networks: a low connectivity and a high connectivity network. The relative performance of the three algorithms was dramatically different in the two cases. In terms of availability, HQC outperformed dynamic voting and majority voting for the low connectivity network with very reliable links, but, overall, dynamic voting resulted in a higher availability. On the other hand, HQC had a uniform advantage over the other two methods in terms of message cost.

In conclusion, HQC is a useful algorithm when the number of objects is large. It is simple, incurs a low message cost, and compares favorably with majority voting in terms of availability.

References


