

Data Synchronization Methods Based on ShuffleNet and Hypercube for Networked Information Systems

David J. Houck*, Kin K. Leung** and Peter Winkler†

* Bell Labs, Lucent Technologies, NJ. Email: dhouck@lucent.com

** Electrical and Electronic Engineering Dept., Imperial College, England. Email: kin.leung@imperial.ac.uk

† Department of Mathematics, Dartmouth College, NH. Email: peter.winkler@dartmouth.edu

Abstract – In contrast to a typical single source of data updates in Internet applications, data files in a networked information system are often distributed, replicated, accessed and updated by multiple nodes. Due to concurrent updates, replicated data files must be synchronized. For certain applications, stringent concurrency control must be employed to ensure data integrity, while for other applications, periodic data synchronization may enable very efficient data sharing. For the latter applications, this paper devises the ShuffleNet and hypercube schemes for data synchronization in such networked information systems. Their performance in terms of update delay, processing complexity, failure tolerance and growth complexity is examined. Our results reveal that the ShuffleNet and hypercube scheme provide identical maximum update delay and similar processing complexity. However, as the number of nodes in the system changes (e.g., due to failure or temporary out of service for maintenance), the hypercube scheme maintains all existing synchronization sessions and greatly simplifies system administration overhead such as moving files from node to node for the purpose of data synchronization. The ShuffleNet scheme does provide a higher degree of failure tolerance for global data files, but the hypercube scheme provides more than adequate failure tolerance. Lastly, a generalization of the hypercube scheme, based on the ideas of shift registers, is also proposed for systems where the number of nodes is a power of 2.

I. INTRODUCTION

In a networked information system, users' data files are often distributed and replicated at multiple, geographically dispersed locations to reduce access time and to improve data availability in case of failure. As a data file can be replicated at multiple locations, these replicated files must be synchronized; otherwise obsolete data can be inadvertently used. For this reason, the issue of concurrency control in such systems has received much attention. See e.g., [1-4] for various concurrency control approaches.

A primitive and perhaps natural method for data synchronization is to allow users to continuously access replicated data stored at their convenient locations and to have the system assume the responsibility of synchronizing the replicated files and reconciling the difference among them *periodically*. This data synchronization approach cannot be

employed in every application. For instance, database records of a banking system, containing the balance for customer accounts, require more stringent concurrency control measure (e.g., locking protocols) to avoid fraudulent activities. On the other hand, for other applications, periodic data synchronization may enable very efficient sharing of data among a group of users such as employees of a large corporation with facilities located at different places. Furthermore, the periodic synchronization method is applicable more widely in distributed computing environments. For example, using this data synchronization, communication and computing services would provide businesses access to a cost-effective client/server computing platform where they could enhance and extend collaboration within their enterprises and among their marketing partners, customers, vendors and suppliers. Other potential applications include network games and distributed simulations.

Let us first define our terminology and outline the system configuration. The networked information system under consideration has multiple *computing nodes* (to be simply referred to as nodes) placed at different locations. Each of these nodes has a database (i.e., a collection of data files) and processing capability. Data files can be categorized as *global data* and *non-global data* in the following way. Global data files are referenced by all users, thus they are replicated and stored at *each* node throughout the system. In contrast, non-global data files are accessed only by a certain group of users (e.g., employees of a company, co-workers of a project) and the files are stored only at selected nodes to meet the needs.

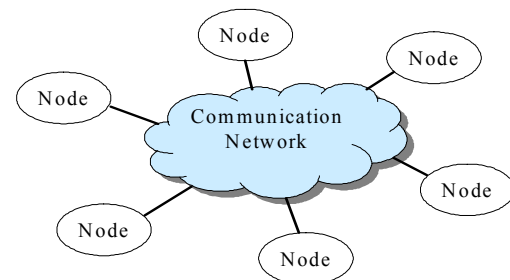


Figure 1. The Networked Information System.

As shown in Figure 1, all nodes can communicate and exchange data with each other over a communication network, which can be a local-area network, wide-area wireless network, etc. When logging onto the system, users are normally connected to a specific node where they expect to find their needed data files in the local database.

A pair of nodes sets up a *synchronization session* to identify and exchange updated copies of their common files. If a common file at only one of the nodes has been changed since the last synchronization, the obsolete copy at the other node is replaced by the new one. If both copies have been updated, the difference between them is stored along with the original version of the file at both nodes. In this case, users are informed of the fact that two copies have been updated concurrently since last synchronization and it is up to them or a system administrator to resolve the update conflict.

Let the *update delay* be the time interval from the instant a file is updated at one node until the updated data becomes available in a replicated copy of the same file at another node. The *maximum update delay* is the longest update delay from any one node to all other nodes. It is important to minimize the maximum update delay by developing an efficient schedule for the synchronization sessions. As data files are expected to be updated continuously, data synchronization must take place repeatedly; the natural approach is to synchronize according to a fixed, periodic schedule.

Since updated data can propagate from one node to another node via some intermediate nodes, the precedence relationship among synchronization sessions may need to be maintained in order to achieve the targeted results. For example, for new data propagated from node A to C via node B, the synchronization session between node A and B should be completed before that between node B and C begins. Thus, it is desirable to complete a synchronization session within the allocated time period with a very high probability. On the other hand, the update delay is directly proportional to the time period allocated for each session. As a result, the time period for each session should be minimized, while keeping the probability of a session exceeding the allotted time satisfactorily low at the engineered traffic load (in terms of update frequency and the amount of data to be exchanged among nodes). In what follows, we assume that the same, fixed time period is allocated to all sessions, regardless of the pair of nodes involved and the amount of data exchanged in the synchronization. The amount of time allocated for each session is thus referred to as *session time*.

If the start of a synchronization session can be triggered by the completion of another, it is possible to guarantee the above precedence relations and to speed up synchronizations for all nodes involved. We ignore this possibility here.

Following are the basic issues in the design of an efficient schedule for synchronization sessions:

- a. What is an appropriate logical topology of nodes? That is, which pairs of nodes should synchronize directly (i.e., update each other without involving a third node)?
- b. How should the sessions be timed so as to optimize performance?
- c. How can the schedule be modified to account for system evolution, such as adding more nodes when user demands grow?
- d. How well does the synchronization (update) schedule perform in the presence of node failures?

An ideal solution to the data synchronization problem should have the following characteristics: a) low maximum update delay, b) ability to withstand at least one node failure, c) adaptability to changes in the number of nodes, and d) minimal transfer of non-global data. (For global data, the amount of data transferred is a constant under any scheme.) With these characteristics in mind, we propose two schemes for data synchronization, which are called the *ShuffleNet scheme* and the *hypercube scheme*.

The rest of this paper is organized as follows. In Section 2, we introduce the ShuffleNet scheme and show its maximum update delay. Since the ShuffleNet scheme requires changes in the update schedule when adding new nodes, this may be unsatisfactory for systems where constant growth is expected. To meet such needs, we propose the hypercube scheme and analyze its performance at normal and failure conditions in Section 3. For a given pair of nodes to share a non-global data file (as specified by user needs), the hypercube scheme may require replicating the file at some intermediate nodes just for the purposes of data synchronization. Section 4 presents a method for identifying the intermediate nodes. We present in Section 5 a generalization of the hypercube scheme that allows each pair of nodes to directly synchronize with each other. Finally, Section 6 presents a summary of performance comparison among a simple pair-wise, the ShuffleNet and the hypercube approaches and our conclusions.

II. THE SHUFFLENET SCHEME

Our ShuffleNet scheme is based on the ShuffleNet topology proposed and studied by Acampora [5], Acampora and Karol [6], and Hluchyj and Karol [7] to enable users to communicate efficiently in multi-hop lightwave networks. In these networks, user nodes communicate with each other possibly via some intermediate nodes (i.e., multi-hop communication), including change of wavelengths in case of wavelength multiplexing on a common optical fiber. Nevertheless, the ShuffleNet provides full connectivity among nodes so that each user can communicate with all other users. This multi-hop communication is analogous to the fact that nodes need to propagate updated data to all other nodes, possibly via some intermediate nodes, in the context of our data synchronization problem.

Our ShuffleNet scheme for data synchronization requires an *even* number $N = 2m$ of nodes in the system, which we

organize into two sets, $X = \{x_0, \dots, x_{m-1}\}$ and $Y = \{y_0, \dots, y_{m-1}\}$; the subscripts are always to be taken modulo m .

Each "round" of communication entails simultaneous synchronization of nodes in X with nodes in Y according to a matching between the two sets. We group the rounds into "batches" of two, batch B_j consisting of rounds R_{2j-1} and R_{2j} for $j \geq 1$.

During each odd batch B_{2j-1} , each x_i synchronizes with $y_{2i+2j-2}$ and with $y_{2i+2j-1}$. It is an easy consequence of Hall's Marriage Theorem (cf. [p.134, 8]) that this can be accomplished with two matchings. Similarly, during even batches B_{2j} , each y_i synchronizes with $x_{2i+2j-2}$ and $x_{2i+2j-1}$. As an example, Figures 2 and 3 present the topology and the processing (update) schedule, respectively, for a system of eight nodes using the ShuffleNet scheme.

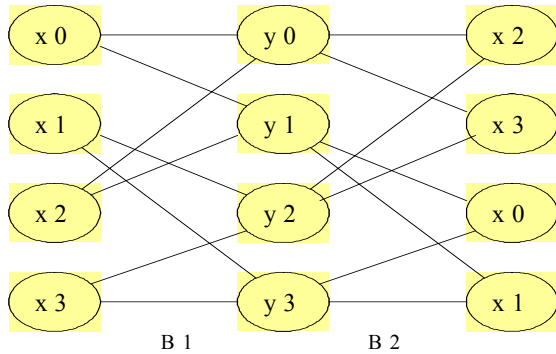


Figure 2. The ShuffleNet Topology for an 8-Node System

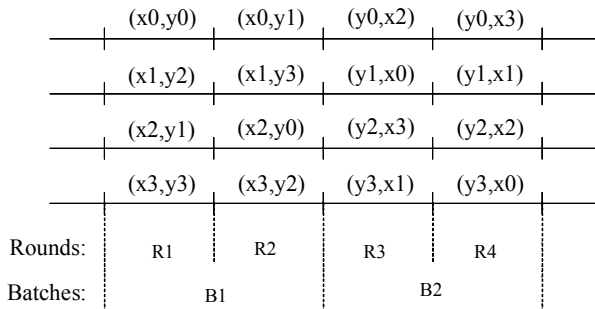


Figure 3. The ShuffleNet Processing Schedule for $N=8$.

It is important to note that in contrast to the original ShuffleNet where connectivity is directional, the edges here are undirected (reflecting the fact that data flows in both directions during a synchronization session). The original ShuffleNet topology also permits more than two node sets and more than two "rounds per batch", but the simple special case utilized here almost always gives the best performance. Also, note that the batch connections are rotated so that our network topology is actually a complete bipartite graph; the reason for this modification will be clear shortly.

Theorem 2.1 *The maximum update delay for the ShuffleNet scheme is at most $2\lceil \log_2 N \rceil + 1$ (where $\lceil x \rceil$ denotes the smallest integer larger than or equal to x in this paper).*

Proof: Let us observe first that if a consecutively-numbered set of k nodes in X have been updated prior to the start of an odd batch, then by the end of the batch either $2k$ consecutive nodes, or all the nodes, of Y have been updated. Of course, a similar statement applies, with the roles of X and Y exchanged, during even batches.

Now, suppose that data is entered at node x_i just prior to round 1; then it reaches all the nodes in one column or the other after $2k$ rounds (k batches) where $k = \lceil \log_2 m \rceil$. One more round will update the nodes in the other column as well, for a total of $2\lceil \log_2 N \rceil - 1$ rounds.

If data is entered at x_i just prior to the *second* round, then the analysis can be applied to its second round mate y_j , at the cost of one round; the case of data entered prior to round 4 is similar. If x_i gets its data just prior to the *third* round, then it and a neighbor (x_{i-1} or x_{i+1}) will be updated by the end of the batch for a total update delay of $2+2\lceil \log_2 m/2 \rceil + 1 = 2\lceil \log_2 N \rceil - 1$ rounds, the same as the first case. The proof is completed by adding a possible delay of at most one round if data is entered just after the start of a round. \square

The maximum update delay can be slightly reduced, asymptotically, by employing three rounds per batch instead of two (on account of the fact that $3^{1/3} > 2^{1/2}$); for larger numbers of rounds per batch, it goes up again. We suspect that in practice the 2-round-per-batch scheme described above will nearly always give the best performance in practice.

Note that our ShuffleNet schedule repeats every m rounds for m even, every $2m$ rounds for m odd; but we could have obtained the same maximum update delay with a schedule which repeats only every four rounds. The reason for rotating as we did, using the complete bipartite graph as our topology, is twofold. First, it then takes $m-1$ node failures to disconnect the network; that is, the rotated schedule is more robust. Secondly, non-global files can be passed *directly* from any X -node to any Y -node, if it becomes desirable to keep them out of the general message flow. From an X to X or Y to Y , just one intermediate node is required.

When new nodes are added to a system, parts of the processing schedule for the ShuffleNet scheme have to be changed accordingly. Thus, the scheme may not be appropriate for systems with constant growth. This has motivated us to devise the hypercube scheme in the following.

III. THE HYPERCUBE SCHEME

Our hypercube scheme for the data synchronization problem is motivated by two observations: a) The data synchronization

problem is closely related to the *gossiping* problem, and b) Nodes in the networked information system can be logically arranged as a hypercube. These observations are discussed in more detail as follows.

The synchronization problem is closely related to the problem of "gossiping in rounds"; see, e.g., the survey by Hedetniemi, Hedetniemi and Liestman [9]. In the original gossiping problem of combinatorial fame, n persons wish to share all they know via a minimal number of telephone calls; it turns out that this minimum number is $2n - 4$, for all $n > 3$.

In the more modern problem of gossiping in *rounds*, disjoint pairs of persons may communicate during a round, and the objective is to minimize the number of rounds necessary for all information to disseminate. In its basic form this problem was solved by Knodel [10], who showed that $\lceil \log_2 N \rceil$ rounds are necessary and sufficient for even N , and $\lceil \log_2 N \rceil + 1$ for odd N . Other work on gossiping in rounds has been done e.g., by Sunderam and Winkler [11], where the case of one-way calls is considered.

As in the problem of gossiping in rounds, our objective in designing a data synchronization schedule is to minimize the number of rounds necessary to disseminate all information among all nodes. However, our problem is "circular" - the schedule is repeated and new data can be introduced at any time, thus the quantity to be minimized is the maximum dissemination time beginning at any point in the schedule. The issues of resistance to node failure and handling of non-global data add further complications, and, in the end, previously proposed gossiping schemes do not serve our purposes as well as those presented here.

Hypercubes have been studied widely by computer scientists and engineers to support efficient communication among multiple processors (see [12]). In particular, hypercube routing and interconnection topology are active research areas. For example, Chiu and Chen [13] study fault-tolerant multicast scheme for hypercube computers, Lai, Chen and Duh [14] investigate disjoint paths in hypercube, and more recently, Fan, Yang and Shiau [15] examine hypercube routing.

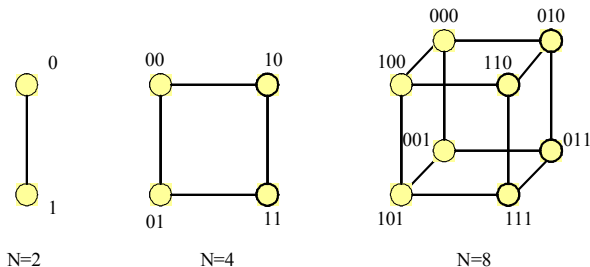


Figure 4. Hypercube for $N=2, 4$ and 8 .

In our hypercube topology, each node is labeled by a binary string. Any two nodes with their labels differing by one bit are connected by an edge and only adjacent nodes (i.e., those

connected by an edge) can communicate and exchange data directly. Yet, all node pairs can communicate perhaps via some intermediate nodes.

To introduce the hypercube scheme for data synchronization, let us begin with simple examples. Figure 4 depicts hypercube for a system with $N = 2, 4$ and 8 nodes. Although the way Figure 4 portrays the connectivity among nodes in a hypercube is common, it is inconvenient for our purposes. Instead, the topology can be more appropriately shown in multiple rounds (or stages) where each round corresponds to the connectivity with node labels differing at a certain bit position. For instance, Figure 5 depicts the 3-round connectivity corresponding to the hypercube for $N = 8$ in Figure 4. In this particular example, edges in round R_2, R_1 and R_0 connect nodes with labels differing in their second, first and zero-th bit (numbered from the right), respectively. In the context of data synchronization, each pair of adjacent nodes in the topology can exchange data directly during a synchronization session at some point in time.

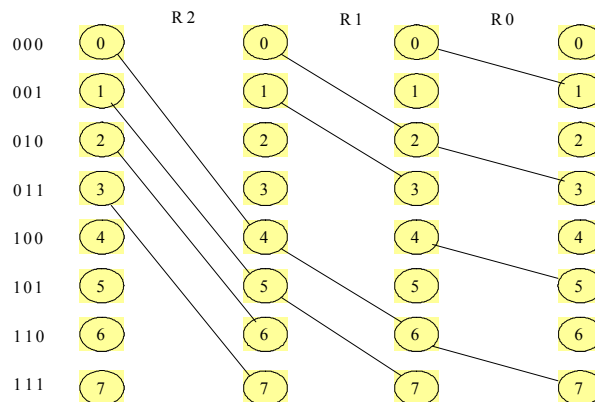


Figure 5. Hypercube Connectivity for $N=8$.

While continuing to use a timer-based processing schedule, one can schedule the synchronization sessions between pairs of nodes to propagate new data from any one node to all other nodes with a maximum update delay of $\log_2 N$ (assuming N is a perfect power of 2) session times. In such a schedule, each round of connectivity also corresponds to one round of synchronization sessions (updates). For the example in Figure 5, the round R_2 of updates allow nodes to synchronize with other nodes if their labels differ in the second bit. Similarly, round R_1 and R_0 enable data exchanges among nodes with labels differing in the first and zero-th bit, respectively. As a result, the processing schedule can be obtained as shown in Figure 6 where (i, j) denotes a synchronization session between node i and j .

It is clear that when N is a perfect power of 2, the hypercube is a useful means to portray the connectivity. However, it is less clear if N is not a perfect power of 2. Solutions developed for the gossiping problem become helpful in understanding and refining the hypercube scheme to handle such general cases.

When $N = 2^k$, Knodel's method for the gossiping problem is essentially identical to the hypercube scheme where nodes are scheduled to exchange information (data) according to the bit position by which node labels differ from each other. Knodel also devised a schedule for even N . However, this schedule requires significant modifications to accommodate the addition of new nodes to the system. Thus, despite its optimality in delay performance, it is not acceptable for our purposes. In contrast, Knodel's solution for odd N simply chooses the first $2^{\lfloor \log_2 N \rfloor}$ nodes and has each of the others synchronize with them in the first and last round of updates, while new data propagates according to the hypercube scheme as discussed above for the $2^{\lfloor \log_2 N \rfloor}$ nodes.

By adopting ideas of the Knodel's method for odd N , the hypercube scheme can now be extended to consider all positive integer N . We first recognize that the last round of updates in the Knodel's method is not needed for the data synchronization problem because the first and last rounds are identical in terms of their data-exchange function and because the processing schedule for data synchronization is repeated continuously. Now, the construction of the logical topology and processing schedule for the hypercube scheme for any positive integer N is described as follows:

- Label each of the N nodes by a binary string where the bits starting from the right are indexed by 0, 1, 2 and so on.
- Establish rounds of the connection topology by connecting nodes with labels differing only in the i -bit position at each round R_i where $i = 0, 1, \dots, \lfloor \log_2 N \rfloor - 1$.
- Establish one round of updates (synchronization sessions) where each pair of adjacent nodes in the corresponding round of connectivity synchronize data.

We show how this "partial hypercube" scheme works below.

A. Correct Update Operations and No Growth Complexity

To see how the hypercube scheme can operate properly and accommodate additional nodes in the system without changing the existing connectivity and processing schedule, let us consider two examples for $N = 6$ and 8. The hypercube connectivity for $N = 8$ and $N = 6$ is shown in Figure 5 and 7, respectively. In addition, their respective processing schedules are depicted in Figure 6 and 8.

The update operations for the case of $N=8$ have been discussed earlier. After three rounds of updates, all nodes can finish exchange of new data among themselves.

As for the case of $N = 6$ in Figure 7, it is noteworthy that the connectivity in round R_1 and R_0 for nodes 0 through 3 actually represents a "perfect" hypercube as if they were the only nodes in the system. Thus, the connectivity in these two rounds enables data exchanges among these four nodes in the same way as for the case of $N = 4$. In addition, the connectivity in round R_2 is to enable data exchanges between the rest of the

nodes with nodes 0 through 3. (This is identical to the first round in Knodel's method for odd N .) Thus, new data from nodes 4 and 5 can be passed onto node 0 to 3 after a first run of all three rounds of updates. Since the processing schedule is repeated continuously (i.e., round R_2 is carried out following round R_0), new data from nodes 0 through 3 can also propagate to nodes 4 and 5 at the second run of round R_2 .

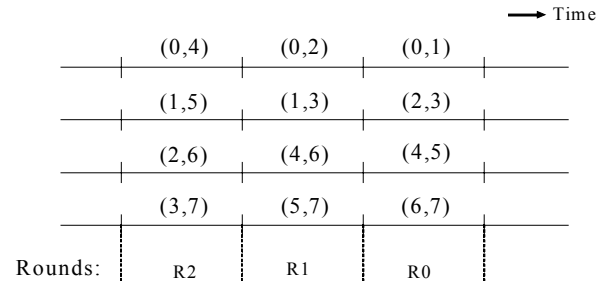


Fig. 6. Processing Schedule for the Hypercube with N=8.

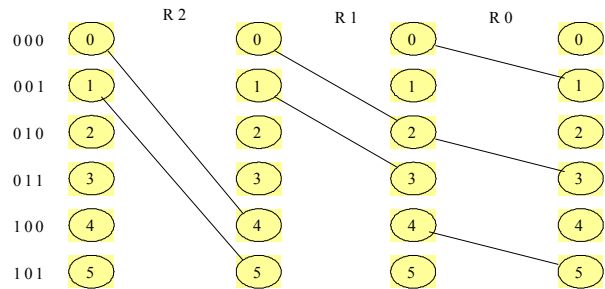


Fig.7 Hypercube Connectivity for N=6.

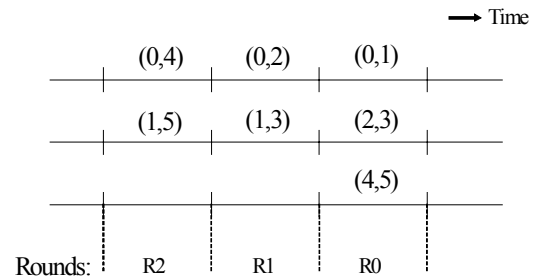


Fig.8 Processing Schedule for the Hypercube Scheme for N=6.

As revealed by Knodel's method, the hypercube scheme with the cyclic processing schedule does not necessarily require the edge between node 4 and 5 in round R_0 to enable data exchanges between node 4 (or 5) and nodes 0 through 3. However, the edge is established by the scheme so that no modifications to the connectivity and thus the processing schedule will be needed when adding new nodes to the system in the future. As a result, it is important to observe that the connectivity for $N = 6$ in Figure 7 is simply a subset of that in Figure 5 for $N = 8$. This lack of need to modify the existing connectivity and processing schedule when systems grow is a unique, desirable characteristic of the hypercube scheme. With this advantage, systems can grow easily by means of adding new nodes to meet increased customer demands.

It is noteworthy that the hypercube scheme remains applicable when nodes are labeled by d -ary strings with $d > 2$. However, the benefits of such arrangements are likely to be realized if d nodes can exchange data simultaneously in a single synchronization session (i.e., a d -way call). So, the scheme with two nodes involved in a session, as presented above, is more practical and adequate to meet our purposes.

B. Delay Performance of the Hypercube Approach

Let the maximum update delay in units of round of updates be denoted by T (where each round of updates lasts for one session time). As pointed out earlier, the hypercube connectivity for a system with N nodes is a subset of that for N' if $N' > N$. We first focus on the maximum update delay for $N=2^m$ and then we consider the delay in a system with arbitrary N nodes.

Theorem 3.1. *Given $N=2^m$ where $m \in \mathbf{Z}^+$, the maximum update delay T is $m+1$ for a global data file in the hypercube scheme, regardless of when the file is actually updated within the (cyclic) processing schedule.*

Proof. Let the source and destination node of an update to a global data file be $S \equiv s_{m-1} s_{m-2} \dots s_0$ and $D \equiv d_{m-1} d_{m-2} \dots d_0$, respectively, where s_i and $d_i \in \{0,1\}$. Further, we denote the rounds of updates in the processing schedule by $R_{m-1}, R_{m-2}, \dots, R_0$. In R_i , each node with label $a_{m-1} \dots a_{i+1} a_i a_{i-1} \dots a_0$ synchronizes with all other node(s) with label $a_{m-1} \dots a_{i+1} \bar{a}_i a_{i-1} \dots a_0$ where $\bar{a}_i = 1 - a_i$.

Without loss of generality, suppose that the data file is updated at S immediately after an arbitrary R_{j+1} starts. This results in one round of delay before the data is ready for propagation in the next round R_j . Then, according to the processing schedule, the new data propagates from node S to node $s_{m-1} s_{m-2} \dots s_{j+1} d_j s_{j-1} \dots s_0$ in R_j . As the processing schedule proceeds through $R_{j-1}, \dots, R_0, R_{m-1}, \dots, R_{j+1}$, the updated data finally reaches the destination node $D = d_{m-1} d_{m-2} \dots d_0$ after a total of m rounds of updates. Thus, $T = m + 1$. \square

To consider the maximum update delay for systems with arbitrary N , let $2^m < N < 2^{m+1}$ and let the nodes also be indexed by $0, 1, \dots, N-1$. Furthermore, define $\Omega \equiv \{0, 1, \dots, 2^m - 1\}$ and $\Sigma \equiv \{2^m, 2^m + 1, \dots, N-1\}$. The maximum update delay for different source and destination node pairs is given as follows.

Theorem 3.2. *For $2^m < N < 2^{m+1}$ with $m \in \mathbf{Z}^+$, the maximum update delay T for different pairs of source node S and destination node D for a global data file in the hypercube scheme is given by*

- a) $m + 2$ if both S and $D \in \Omega$,
- b) $2(m + 1)$ if $S \in \Omega$ and $D \in \Sigma$,
- c) $2(m + 1)$ if $S \in \Sigma$ and $D \in \Omega$, and

d) $2m + 3$ if $S \in \Sigma$ and $D \in \Sigma$.

Proof. As in Theorem 3.1, the rounds of updates in the cyclic processing schedule are denoted by R_m, R_{m-1}, \dots, R_0 . In R_i for $i=m, m-1, \dots, 0$, each node with label $a_{m-1} \dots a_{i+1} a_i a_{i-1} \dots a_0$ synchronizes with all other node(s) with label $a_{m-1} \dots a_{i+1} \bar{a}_i a_{i-1} \dots a_0$ where $\bar{a}_i \in \{0, 1\} - \{a_i\}$.

By definition of the hypercube scheme, all nodes in Ω actually form a perfect hypercube. Thus, results for case a) are obvious from Theorem 3.1 because all rounds of updates, namely R_{m-1}, R_{m-2}, \dots , and R_0 , needed to complete data exchanges among all nodes in Ω have been processed after $m + 1$ rounds of updates (only R_m is considered unproductive). Beside this delay, up to one additional round of delay can incur if the update arrives immediately after the start of a round.

To consider case b), recall that a node in Σ synchronizes with its mate node in Ω (the one with the m 'th bit set to one.) only during R_m . It may synchronize with other nodes in Σ , but in the worst case a destination node $D \in \Sigma$ must receive the updated data from its mate node \bar{D} and only during R_m . Furthermore, in order for node D to receive the new data, the data must have already propagated from the source node $S \in \Omega$ to \bar{D} prior to R_m . Since S and \bar{D} may differ in bit $m-1$, the longest delay occurs when the associated data file was updated at S immediately after the start of R_{m-1} . As a result, the new data will not propagate to \bar{D} in the following m rounds of updates, namely $R_{m-1}, R_{m-2}, \dots, R_0$. Consequently, the time interval for the m rounds plus the following R_m is wasted as far as the propagation of the new data is concerned. However, by the arguments for Theorem 3.1, the new data will reach all nodes in Ω , including \bar{D} , in the second run of the m rounds, R_{m-1}, R_{m-2}, \dots , and R_0 . Finally, the destination node D receives the updated data from during the second run of R_m . Thus, the maximum update delay for case b) is $2(m + 1)$.

Following the same arguments for case b), the maximum delay for case c) occurs when the data file is updated at a source node S in Σ immediately after the start of R_m . In this situation, the new data cannot be passed from S to its mate \bar{S} in Ω during R_m . As a result, the time interval for R_m and the following m rounds, R_{m-1}, R_{m-2}, \dots , and R_0 , is wasted as far as the data propagation from S to D is concerned. This accounts for a total of $m+1$ rounds of updates. At the second run of R_m , the new data is finally passed from S to \bar{S} . By the arguments for Theorem 3.1, the destination node D in Ω definitely receives the updated data in the next m rounds of updates. Thus, the total delay is again $2(m + 1)$.

For case d), the longest update delay occurs when the new data has to propagate from $S \in \Sigma$ to $D \in \Sigma$ via some intermediate

nodes in Ω . This is so because the nodes in Σ may not form a connected graph. Following the above arguments, the maximum delay is realized when the data file is updated immediately after the start of R_m . Since the new data cannot be passed from S to any node in Ω , the following $m+1$ rounds of updates (i.e., R_m, R_{m-1}, \dots, R_0) are wasted for the data propagation from S to D . In the second run of the schedule, starting with R_m and R_{m-1} through R_0 , the new data reaches all nodes in Ω by the end of R_0 . Then, in the third run of R_m , the new data is passed from node \bar{D} in Ω to node D in Σ . Thus, the total delay in units of rounds is $2(m+1)+1$ for case d). \square

C. Processing Complexity

We first analyze the number of sessions processed in each round of updates in the schedule. Then, the processing complexity for the hypercube scheme can be obtained in terms of the total number of synchronization sessions processed for the longest update path described in Theorems 3.1 and 3.2.

Let each of the N nodes in the system have a label expressed in binary form. Furthermore, let $N-1 = (n_m n_{m-1} \dots n_0)_2$ where $n_i = 0$ or 1 . For $0 \leq i \leq m$, define K_i to be the total number of synchronization sessions processed at round R_i .

Lemma 3.1. For $0 \leq i \leq m$,

$$K_i = \begin{cases} \sum_{j=i}^{m-1} 2^j n_{j+1} & \text{if } n_i = 0 \\ \sum_{j=i}^{m-1} 2^j n_{j+1} + \sum_{j=0}^{i-1} 2^j n_{j+1} & \text{if } n_i = 1. \end{cases} \quad (3.1)$$

Proof. Consider $n_i = 0$. At round R_i , all nodes with labels from 0 to $(n_m n_{m-1} \dots n_{i+1} 0 \dots 0)_2 - 1$ are involved in synchronization sessions. This is so because all these nodes can find their counterpart nodes with labels differing in the i^{th} bit (i.e., the bit position of n_i). In total, there are $\sum_{j=i+1}^m 2^j n_j$ nodes. On the other hand, nodes with labels from $(n_m n_{m-1} \dots n_{i+1} n_i 0 \dots 0)_2$ to $(n_m n_{m-1} \dots n_{i+1} n_i \dots n_0)_2$ cannot find their counterpart nodes to synchronize data at round R_i because they do not exist as $n_i = 0$.

Changing the index in the summation $\sum_{j=i+1}^m 2^j n_j$ and observing that two nodes are involved in each synchronization session yield the result.

For $n_i = 1$, the $\sum_{j=i+1}^m 2^j n_j$ nodes continue to find their counterpart nodes. In addition, as $n_i = 1$, some nodes with labels from $(n_m n_{m-1} \dots n_{i+1} 0 \dots 0)_2$ to $(n_m n_{m-1} \dots n_{i+1} 01 \dots 1)_2$ can also be involved in synchronization sessions at R_i with some nodes with labels ranging from $(n_m n_{m-1} \dots n_{i+1} 10 \dots 0)_2$ to $(n_m n_{m-1} \dots n_{i+1} 1 n_{i-1} \dots n_0)_2$ as long as these counterpart nodes exist. The proof is completed because the number of these counterpart nodes existing in the system is given by $\sum_{j=0}^{i-1} 2^j n_{j+1} + 1$. \square

The processing complexity of the hypercube scheme is defined as the number of synchronization sessions processed during the maximum update delay for any node pair. Theorem 3.2, case d) yields the maximum update delay. Thus, using results in Lemma 3.1, the processing complexity is obtained.

Theorem 3.3. The processing complexity for the hypercube scheme is $2 \sum_{i=0}^m K_i + K_m$ where K_i for $0 \leq i \leq m$ is given by (3.1).

D. Performance Under Node Failures

In the following, we first obtain the degree of failure tolerance as well as the increase in update delay under failure for the case where N is a perfect power of 2. Then we show the failure tolerance for the case of general N for the hypercube scheme.

For $N = 2^m$, the following theorem shows how the update delay is impacted by at most one node failures.

Theorem 3.4. For the hypercube scheme with $N=2^m$ for $m \in \mathbf{Z}^+$, if one node fails, the maximum update delay T in units of update round is at most $m+2$ (i.e., one more round than normal) for any global data file.

Proof. Let an arbitrary source and destination node of an update for a global data file be $S \equiv s_{m-1} s_{m-2} \dots s_0$ and $D \equiv d_{m-1} d_{m-2} \dots d_0$ respectively, where s_i and $d_i \in \{0, 1\}$.

Let the file be updated at S right after the start of R_{j+1} , thus resulting in one round of delay until the updated data can be passed to other nodes starting at round R_j . Then, by the cyclic processing schedule of the hypercube scheme, the shortest (update) path to propagate updated data from S to D without node failure consists of the following intermediate nodes:

Round	$s_{m-1} s_{m-2} \dots s_{j+1} s_j s_{j-1} \dots s_0$
R_j	$s_{m-1} s_{m-2} \dots s_{j+1} d_j s_{j-1} \dots s_0$
R_{j-1}	$s_{m-1} s_{m-2} \dots s_{j+1} d_j d_{j-1} \dots s_0$
..	..
..	..
R_0	$s_{m-1} s_{m-2} \dots s_{j+1} d_j d_{j-1} \dots d_0$
R_{m-1}	$d_{m-1} s_{m-2} \dots s_{j+1} d_j d_{j-1} \dots d_0$
..	..
..	..
R_{j+2}	$d_{m-1} d_{m-2} \dots s_{j+1} d_j d_{j-1} \dots d_0$
R_{j+1}	$d_{m-1} d_{m-2} \dots d_{j+1} d_j d_{j-1} \dots d_0$

Table 1. Update Path From S to D in Normal Situations.

Let U be the set of these intermediate nodes, excluding S and D . Further, assume that set F is the failed node. If $U \cap F = \emptyset$, the above update path is still operational and the update delay T

remains $m+1$. For $U \cap F \neq \emptyset$, consider the following set of update paths from S to D :

Round	$s_{m-1} s_{m-2} \dots s_{j+1} s_j^s s_{j-1} \dots s_0$
R_j	$s_{m-1} s_{m-2} \dots s_{j+1} \bar{d}_j^s s_{j-1} \dots s_0$
R_{j-1}	$s_{m-1} s_{m-2} \dots s_{j+1} \bar{d}_j^d s_{j-1} \dots s_0$
..	..
..	..
R_0	$s_{m-1} s_{m-2} \dots s_{j+1} \bar{d}_j^d s_{j-1} \dots d_0$
R_{m-1}	$d_{m-1} s_{m-2} \dots s_{j+1} \bar{d}_j^d s_{j-1} \dots d_0$
..	..
..	..
R_{j+2}	$d_{m-1} d_{m-2} \dots s_{j+1} \bar{d}_j^d s_{j-1} \dots d_0$
R_{j+1}	$d_{m-1} d_{m-2} \dots d_{j+1} \bar{d}_j^d s_{j-1} \dots d_0$
R_j	$d_{m-1} d_{m-2} \dots d_{j+1} d_j^d s_{j-1} \dots d_0$

Table 2. Update Path from S to D with at Most One Node Fails

Where $\bar{d}_j = 1 - d_j$. This set of nodes describes another update path. Let the set of intermediate nodes (excluding S and D) in this path be denoted by V . By construction, the update path with V consists of $m+1$ rounds of updates. Combining this with the round wasted at which the update arrives, the update delay T is $m+2$ rounds. Furthermore, we have $U \cap V = \emptyset$ simply because $\bar{d}_j \neq d_j$. Consequently, U and V represent two disjoint update paths from S to D . Given that at most one node fails, at least one of the paths remains unaffected by the failures and the delay is at most $m+2$. \square

The hypercube scheme actually can tolerate more than one node failure, but the maximum update delay is further increased. The result is presented in the following theorem.

Theorem 3.5. *For the hypercube scheme with $N=2^m$ for $m, k \in \mathbf{Z}^+$ and $k \leq m-1$, if no more than k nodes fail, then all data will propagate and the maximum update delay T is at most $m+k+2$ (i.e., $k+1$ more rounds than normal) for any global data file.*

Proof. As before, let the cyclic processing schedule for the hypercube scheme be $R_{m-1}, R_{m-2}, \dots, R_0$, where nodes in R_i synchronize data with other nodes with labels differing at the i -th bit position. Let an update to a global data file be made at arbitrary node $S \equiv s_{m-1} s_{m-2} \dots s_0$ just after R_{j+1} starts. After a one-round delay, the updated data is ready to be propagated in R_j . We need to show the updated data can reach any arbitrary destination $D \equiv d_{m-1} d_{m-2} \dots d_0$ in the following $m+k+1$ rounds. Under normal conditions, the update can propagate along the shortest path P from S to D described in Table 1. Let U be the set of intermediate nodes in the path. Let F be the set of failed nodes. If $F \cap U = \emptyset$, the path P is still operational and the update delay T remains $m+1$, as proven in Theorem 3.1.

Now, consider $F \cap U \neq \emptyset$. Given $k \leq m-1$, we can always construct $k+1$ update paths, denoted by P_i where $i = j, j-1, \dots, j-k$ as follows. (Possibly, $i = j, j-1, \dots, 0, m-1, \dots, m+j-k$ for small j or large k .) Each path P_i is obtained by purposely delaying the propagation of the updated data from S for $(j-i+m) \bmod m$ rounds starting from R_j , and then passing the data at the next round R_i from S to the node with label where the i -th bit is $\bar{d}_i = 1 - d_i$. The data continues to propagate in the next $m-1$ successive rounds starting from R_{i-1} (or R_{m-1} if $i=0$) to nodes with the bits associated with the rounds identical to the corresponding ones in D . Finally, when R_i comes again, the updated data is passed to D from the node with the i -th bit being d_i . For example, P_j and P_{j-1} are:

Round	$s_{m-1} s_{m-2} \dots s_{j+1} s_j^s s_{j-1} \dots s_0$
R_j	$s_{m-1} s_{m-2} \dots s_{j+1} \bar{d}_j^s s_{j-1} \dots s_0$
R_{j-1}	$s_{m-1} s_{m-2} \dots s_{j+1} \bar{d}_j^d s_{j-1} \dots s_0$
..	..
..	..
R_0	$s_{m-1} s_{m-2} \dots s_{j+1} \bar{d}_j^d s_{j-1} \dots d_0$
R_{m-1}	$d_{m-1} s_{m-2} \dots s_{j+1} \bar{d}_j^d s_{j-1} \dots d_0$
..	..
..	..
R_{j+2}	$d_{m-1} d_{m-2} \dots s_{j+1} \bar{d}_j^d s_{j-1} \dots d_0$
R_{j+1}	$d_{m-1} d_{m-2} \dots d_{j+1} \bar{d}_j^d s_{j-1} \dots d_0$
R_j	$d_{m-1} d_{m-2} \dots d_{j+1} d_j^d s_{j-1} \dots d_0$

Table 3. Update Path P_j

Round	$s_{m-1} s_{m-2} \dots s_{j+1} s_j^s s_{j-1} s_{j-2} \dots s_0$
R_j	$s_{m-1} s_{m-2} \dots s_{j+1} s_j^s s_{j-1} s_{j-2} \dots s_0$
R_{j-1}	$s_{m-1} s_{m-2} \dots s_{j+1} s_j \bar{d}_j^s s_{j-1} s_{j-2} \dots s_0$
R_{j-2}	$s_{m-1} s_{m-2} \dots s_{j+1} s_j \bar{d}_j^d s_{j-1} d_{j-2} \dots s_0$
..	..
R_0	$s_{m-1} s_{m-2} \dots s_{j+1} s_j \bar{d}_j^d s_{j-1} d_{j-2} \dots d_0$
R_{m-1}	$d_{m-1} s_{m-2} \dots s_{j+1} s_j \bar{d}_j^d s_{j-1} d_{j-2} \dots d_0$
..	..
R_{j+2}	$d_{m-1} d_{m-2} \dots s_{j+1} s_j \bar{d}_j^d s_{j-1} d_{j-2} \dots d_0$
R_{j+1}	$d_{m-1} d_{m-2} \dots d_{j+1} s_j \bar{d}_j^d s_{j-1} d_{j-2} \dots d_0$
R_j	$d_{m-1} d_{m-2} \dots d_{j+1} d_j \bar{d}_j^d s_{j-1} d_{j-2} \dots d_0$
R_{j-1}	$d_{m-1} d_{m-2} \dots d_{j+1} d_j d_{j-1} d_{j-2} \dots d_0$

Table 4. Update Path P_{j-1}

respectively. By construction, each path P_i has a delay of $(j-i+m) \bmod m + (m+1)$ rounds for each $i = j, j-1, \dots, j-k$ (or $i = j, j-1, \dots, 0, m-1, \dots, m+j-k$ for small j or large k). Including

the initial waiting in R_{j+1} when the update arrives, the longest update delay among all P_i 's is thus $m+k+2$ update rounds.

For each path P_i , let V_i be the set of intermediate nodes (excluding S and D). We now show that $V_a \cap V_b = \emptyset$ for all a and b ($a \neq b$) by contradiction.

Assume that $V_a \cap V_b \neq \emptyset$ for some a and b . Thus, there exist at least one node $\mathbf{x} \in V_a$ and one node $\mathbf{y} \in V_b$ such that $\mathbf{x} = \mathbf{y}$. Neither \mathbf{x} or \mathbf{y} are S or D . Further, assume that \mathbf{x} is the intermediate node at round R_u in path P_a and \mathbf{y} is from round R_v in P_b . By the path construction, we have

$$\begin{aligned} \mathbf{x} &= s_{m-1} \dots s_{a+1} \bar{d}_a d_{a-1} \dots d_u s_{u-1} \dots s_0 \\ \mathbf{y} &= s_{m-1} \dots s_{b+1} \bar{d}_b d_{b-1} \dots d_v s_{v-1} \dots s_0 \end{aligned} \quad (3.2)$$

We also define $\mathbf{x} \equiv x_{m-1} x_{m-2} \dots x_0$ and $\mathbf{y} \equiv y_{m-1} y_{m-2} \dots y_0$. Thus, $\mathbf{x} = \mathbf{y}$ requires $x_i = y_i$ for all i . To proceed, let us define $M \equiv \{m-1, m-2, \dots, 0\}$, $I_x \equiv \{a, a-1, \dots, u\} \subseteq M$ and $I_y \equiv \{b, b-1, \dots, v\} \subseteq M$. Another way to express (3.2) is

$$\begin{aligned} x_a = \bar{d}_a, x_i = d_i \text{ for } i \in I_x - \{a\} \\ \text{and } x_i = s_i \text{ for } i \in M - I_x \end{aligned} \quad (3.3)$$

$$\begin{aligned} y_b = \bar{d}_b, y_i = d_i \text{ for } i \in I_y - \{b\} \\ \text{and } y_i = s_i \text{ for } i \in M - I_y \end{aligned} \quad (3.4)$$

If $b \in I_x - \{a\}$, $x_b = d_b$ by (3.3). However, $y_b = \bar{d}_b$ from (3.4). This leads to $\mathbf{x} \neq \mathbf{y}$. So, we must have $b \notin I_x$. Similarly, if $a \in I_y - \{b\}$, $x_a = d_a$ and $y_a = \bar{d}_a$ leads to $\mathbf{x} \neq \mathbf{y}$. Thus, we must also have $a \notin I_y$.

Since both I_x and I_y are sets of a decreasing sequence of modulo- m integers starting with a and b , respectively, the fact that $a \notin I_y$ and $b \notin I_x$ implies $I_x \cap I_y = \emptyset$. Since $I_x \subseteq M$ and $I_y \subseteq M$, $I_x \cap I_y = \emptyset$ further implies $M - I_x \supseteq I_y$ and $M - I_y \supseteq I_x$. Using this fact, we obtain from (3.3) that $x_i = s_i$ for all $i \in M - I_x \supseteq I_y$. This implies $y_i = s_i$ for all $i \in I_y$ because $x_i = y_i$ for all $i \in M$. Combining this with the case of y_i for $i \in M - I_y$ in (3.4) leads to $y_i = s_i$ for all $i \in M$. That is, $\mathbf{y} = S$, a contradiction. \square

Define the *degree of failure tolerance* to be the maximum number of node failures under which each remaining node can still exchange data, directly or indirectly via some intermediate nodes, with all other operational nodes. By setting $k = m - 1$ in Theorem 3.5, we get the following result.

Corollary 3.1. *For the hypercube scheme with $N=2^m$, the degree of failure tolerance is at least $m-1$.*

When N is not a perfect power of 2, we now derive the connectivity of the partial hypercube and thus the degree of failure tolerance. We also describe the impact on update delay here.

Theorem 3.6. *For the hypercube with $2^m < N \leq 2^{m+1}$ with $N = (n_m n_{m-1} \dots n_0)_2 + 1$, the degree of failure tolerance is given by*

$$f = \sum_{i=0}^m n_i - 1 \quad (3.5)$$

Proof:

Case 1) If $N = 2^{m+1}$ then $\sum_{i=0}^m n_i = (m+1)$ and the result is true by Corollary 3.1.

Case 2) Assume that $N < 2^{m+1}$. In this case $\sum_{i=0}^m n_i \leq m$.

We prove the result by induction on m . For $m = 1$, we have $N=3$ and the nodes are disconnected if node 0 fails. Thus, the result is correct. Assume the theorem is true for $N < 2^m$. We shall prove that it is true for $N < 2^{m+1}$.

Define Ω to be the set of nodes with bit m equal to 0 and Σ the set of nodes with bit m equal to 1. Let S and D be an arbitrary source and destination as before. We shall show that there exist $f+1$ node disjoint paths between S and D . This is equivalent to showing the partial hypercube is $f+1$ -connected.

Case 2A) $S, D \in \Omega$: Ignoring the m 'th bit (which is 0 for all nodes in Ω), it can be seen that Ω is a perfect hypercube of size d^m . By Case 1, there exist m disjoint paths within Ω between S and D . Since $m \geq \sum_{i=0}^m n_i = f+1$, this case is shown.

Case 2B) $S, D \in \Sigma$: In this case, when ignoring bit m , we see that Σ is a partial hypercube with $N' < d^m$. By the inductive step, there exist $\sum_{i=0}^{m-1} n_i = f$ disjoint paths between S and D within Σ . Let S_0 and D_0 denote the nodes in Ω where bit m of S and D are set to 0. Since Ω is a connected subgraph there is a path from S_0 to D_0 within Ω . This gives us one additional disjoint path and proves this case.

Case 2C) $S \in \Sigma$ and $D \in \Omega$: There exists a matching of size $f+1$ between nodes in Σ and those in Ω . Simply pick any $f+1$ nodes in Σ . Changing bit m from 1 to 0 connects each of these nodes to a unique node in Ω . If any f points are removed from the total hypercube, then there still must be a matched pair of nodes between Σ and Ω , call them a and b respectively. Now cases 2A and 2B above showed that any two nodes in Σ or two nodes in Ω are connected by $f+1$ disjoint paths. Thus there still exists a path remaining between S and a , and between b and D . Therefore, there still exists a path from S to D and the hypercube is $f+1$ -connected.

Case 2D) $S \in \Omega$ and $D \in \Sigma$: Same proof as case 2C. \square

E. Identifying Intermediate Nodes for Non-Global Data Files

In many cases, a copy of a non-global data file must be replicated on an intermediate node to enable new data associated with a file to propagate via that node. Let $2^{m+1} < N \leq 2^m$ and nodes be labeled from 0 to $N-1$ expressed in binary form. When users at two arbitrary nodes $S \equiv s_m s_{m-1} \dots s_0$ and $D \equiv d_m d_{m-1} \dots d_0$ share a non-global data file, the system administrator may need to replicate the file on a set of intermediate nodes for the purpose of data synchronization. There are two competing objectives at play here. One is to minimize the number of intermediate nodes at which this data is replicated and two is to minimize the update delay. We discuss two methods to identify these intermediate nodes as follows.

In the first method, we emphasize the delay criteria, by finding the shortest path from S to D and the shortest path from D to S . Although a generic shortest path algorithm could be used, the following paths are usually acceptable. By propagating data through a sequence of nodes with one-bit difference at each round starting from S in round R_m , we obtain a set U of the intermediate nodes. Namely, $U = \{ d_m s_{m-1} s_{m-2} \dots s_0, d_m d_{m-1} s_{m-2} \dots s_0, \dots, d_m d_{m-1} d_{m-2} \dots d_0 \}$. If all nodes in U exist in the system, U represents a shortest update path from S to D . Otherwise, one can always find a feasible path with $U = \{ 0 s_{m-1} s_{m-2} \dots s_0, 0 d_{m-1} s_{m-2} \dots s_0, \dots, 0 d_{m-1} d_{m-2} \dots d_0, d_m d_{m-1} d_{m-2} \dots d_0 \}$ as those nodes with the left-most bit being zero always exist in the system. Similar to the update path in Table 1, the intermediate nodes in U follow the sequence of the processing schedule, thus yielding low delay to propagate data from S to D .

Since data propagates bidirectionally between each pair of nodes involved in a synchronization session, U in principle also enables data propagation from D to S . However, such a path yields a long delay because U does not follow the sequence of nodes to support efficient data propagation from D to S . To remedy this and to enhance the failure tolerance for non-global data files, one can apply the above method to identify another set V of intermediate nodes for data propagation from D to S . When N is a perfect power of 2, U and V actually represent two node-disjoint paths.

We note that the shortest paths between S and D may depend on the number of nodes in the system at the time of creation of the non-global file. If the number of nodes changes then there could be different update paths U and V for different files even though they are shared by the same pair of users at two fixed nodes. If the system records the intermediate nodes for each non-global data file for maintenance, failure recovery and other system administration purposes, one can make the current value of N as a *permanent attribute* for each file when the method is applied to find the corresponding intermediate nodes. Once N is fixed and known for a non-global file, its U and V can be uniquely and readily obtained at a later time.

The second method is needed, if the system requires that all non-global data files shared by the same pair of users have identical intermediate nodes, despite of constant changes of the number of nodes in the system. To identify the paths U and V for the given S and D in this case, the second method finds the maximum of S and D . Without loss of generality, let D be the maximum. Then, the second method follows the first one, except that the number of nodes is deliberately assumed to be D (instead of N) when identifying U and V . This way, all intermediate nodes for the given S and D are kept identical all the times at the expense of possibly longer update delay because part of the connectivity in the hypercube topology (i.e., that among nodes $D+1$ to $N-1$) is not utilized for possibly more efficient data propagation.

IV. A GENERALIZATION OF THE HYPERCUBE SCHEME

We present a generalization of the hypercube scheme for the number of nodes being a perfect power of 2 (i.e., $N = 2^m$). The generalized scheme again employs the labeling of nodes by their binary representation, but the matchings used to determine the update schedule will no longer be confined to hypercube edges. Instead, if $x = (x_1, \dots, x_m)$ is a non-zero binary vector, we denote by M_x the matching which for each u , pairs the node labeled u with the node labeled $x + u$. (Here addition is coordinate-wise and modulo 2, so that $x + u + u = x$ and therefore the pairing relation is symmetric.)

The matchings employed by the standard hypercube solution to gossiping in rounds are just M_x for $x = 000\dots010\dots0$.

If we cycle through matchings corresponding to all the non-zero vectors in some order, say $x(1), x(2), \dots, x(n-1)$ and back to $x(1)$, each node is paired with every other; in other words, we achieve the desired complete topology. But the maximum update delay will generally be larger than m . Fortunately, optimality of the update delay turns out to be equivalent to a simple condition.

Theorem 4.1. *The scheme which cycles through matchings $M_{x(1)}, \dots, M_{x(n-1)}$ where the $x(i)$'s run through all the non-zero binary vectors of length m , achieves maximum update delay m if and only if for every i , the vectors $x(i), x(i+1), \dots, x(i+m-1)$ (with indices taken modulo $n-1$) are linearly independent over $GF(2)$.*

Proof. To see that the condition is sufficient, suppose node u receives a new update just prior to round i and let v be any other node. If $x(i), \dots, x(i+m-1)$ are linearly independent they constitute a basis for $GF(2)^m$, thus $v - u = x(i_1) + x(i_2) + \dots + x(i_k)$ for some i_1, \dots, i_k with $i \leq i_1 < i_2 < \dots < i_k \leq i+m-1$. But then the path $u, u+x(i_1), u+x(i_1)+x(i_2), \dots$ takes the update information from u to v as required.

On the other hand, if $x(i), \dots, x(i+m-1)$ are not linearly independent then they do not span $\text{GF}(2)^m$ and there is a vector y which is not expressible as a linear combination. There is then no path from u to $u+y$ during the upcoming m rounds. \square

In view of Theorem 4.1, we are in need of a listing of the non-zero members of $\text{GF}(2)^m$ in which any m in a row (including "around the corner") are independent. We can get it by starting anywhere and applying a "full-cycle shift register".

An order- m linear shift register is a sequence of m boxes b_1, \dots, b_m , each of which contains a bit, and some of which are "tapped." At each tick of a clock the bits move to the right one box, the rightmost bit dropping off into oblivion. The new leftmost bit is the mod-2 sum of the old bits in the tapped boxes.

Suppose the tapped boxes are b_2, b_3 and b_5 . Then if the current state is $x = x_1, \dots, x_m$, the next state will be $y = y_1, \dots, y_m$ where $y_{i+1} = x_i$ and $y_1 = x_2 + x_3 + x_5$.

More generally, if we define $c_i=1$, if b_i is tapped and $c_i = 0$ otherwise, and we denote by $x(t) = x_1(t), \dots, x_m(t)$ the contents of the shift register at time t , we have

$$x_{i+1}(t+1) = x_i(t) \text{ for } i+1=2, \dots, m \text{ and} \\ x_1(t+1) = c_1 x_1(t) + \dots + c_m x_m(t).$$

In fact, if we just look at the first box, its contents $x_1(t)$ satisfy the recursion

$$x_1(t) = c_1 x_1(t-1) + c_2 x_1(t-2) + \dots + c_m x_1(t-m)$$

since $x_1(t-i)$ is the same as $x_{1+i}(t)$. In fact every box satisfies the same recursion, so we can write a vector equation

$$x(t) = c_1 x(t-1) + \dots + c_m x(t-m).$$

If we start a shift register with all zeroes it goes nowhere, but one can hope that otherwise it will cycle through all non-zero vectors before it returns to its starting state. The theory of shift registers (see e.g., [16]) tells us exactly when this happens.

Let $f(z)$ be the polynomial $1 + c_1 z + c_2 z^2 + \dots + c_m z^m$. Then the shift register is "full-cycle" just when $f(z)$ is irreducible over $\text{GF}(2)$ and its smallest multiple of the form $1+z^k$ is for $k=2^m-1$.

Fortunately there are plenty of polynomials with this property, in fact exactly q/m of them where q is the number of proper irreducible fractions of the form $i/(2^m-1)$. Examples:

$$m=2, f(z) = 1 + z + z^2, c = 11;$$

$$m=3, f(z) = 1 + z + z^3, c = 101;$$

$$m=4, f(z) = 1 + z + z^3 + z^5, c = 0011;$$

$$m=5, f(z) = 1 + z + z^3 + z^5, c = 00101.$$

Here are the resulting vector sequences, starting with 00...01:
 $m=2$: 01, 10, 11, 01, ...

$m=3$: 001, 100, 110, 111, 011, 101, 010, 001, ...

$m=4$: 0001, 1000, 0100, 0010, 1001, 1100, 0110, 1011, 0101, 1010, 1101, 1110, 1111, 0111, 0011, 0001, ...

Now the optimal update delay comes for free, compliments of the vector equation above which says that each vector is a linear combination of the previous m vectors. If any m -in-a-row failed to be independent then all subsequent vectors (therefore all vectors, since we cycle around) would be limited to the subspace spanned by those m vectors, but in fact all non-zero vectors appear.

What about resistance to a single node failure? The classic hypercube structure had the nice property that maximum update delay increases by only 1 in that case (best possible); interestingly, that nice property also follows from m -in-a-row independence. Suppose that node u is updated just prior to round $t-1$, with the idea of showing that there are two node-disjoint paths from u to v during the next $m+1$ rounds. We know there is a path P from u to v during rounds t through $t+m-1$, and a path Q from $u+x(t-1)$ to v during the same period. Suppose P and Q have a node in common other than v itself; then we have

$$u + x(i_1) + x(i_2) + \dots + x(i_r) = u + x(t-1) + x(j_1) + \dots + x(j_s)$$

where all of the subscripted i 's and j 's lie between t and $t+m-2$. But then we have $x(t-1)$ expressed as a linear combination of the $m-2$ vectors $x(t), x(t+1), \dots, x(t+m-2)$, contradicting the fact that $x(t-1)$ through $x(t+m-2)$ are supposed to be independent.

The case of multiple node failures is more complex, but since the topology is complete, any number of node failures is overcome via update delay of no more than $N-1$ rounds.

Since the topology is complete, this generalization has the potentially significant advantage of allowing non-global data files to be updated without any intermediate nodes involved. Of course, there will be an increase in update delay.

V. PERFORMANCE COMPARISON AND CONCLUSION

We have devised the ShuffleNet and hypercube scheme for data synchronization in the networked information systems with replicated data files. Their performance in terms of update delay, processing complexity (i.e., number of synchronization sessions in the longest update path), failure tolerance and growth complexity has been examined. For illustration purposes, Table 5 compares the performance of these schemes and a simple, pair-wise approach where each node synchronizes directly with each of all other nodes one at a time in the schedule. It is found that the ShuffleNet and hypercube scheme provide identical maximum update delay and similar processing complexity. However, as the number of nodes in the system changes, the hypercube scheme maintains all existing synchronization sessions and greatly simplifies system administration overhead such as moving files from node to node for the purpose of non-global data synchronization. The ShuffleNet scheme does provide a higher

degree of failure tolerance for global data files, but the hypercube scheme provides more than adequate failure tolerance. Lastly, the generalization of the hypercube scheme, based on the ideas of shift registers, is also proposed for systems where the number of nodes is a perfect power of 2.

Performance Measure	Simple, Pair-wise Approach	ShuffleNet Approach	Hypercube Approach
Max. update delay (session times)	$\frac{N(N-1)}{2}$	$2\lceil \log_2 N \rceil + 1$	$2\lceil \log_2 N \rceil + 1$
Number of synchronization sessions in the longest path	$\frac{N(N-1)}{2}$	$N(\lceil \log_2 N \rceil + \frac{1}{2})$	$\leq N(\lceil \log_2 N \rceil + \frac{1}{2})$
Degree of failure tolerance	$N - 2$	$\frac{N}{2} - 2$	$\leq \lceil \log_2 N \rceil - 1$
Restriction on N	No	Even N	No
Growth complexity	No	Some	No

Note: $\lceil x \rceil$ is the smallest integer larger than or equal to x .

TABLE 5. PERFORMANCE COMPARISON

ACKNOWLEDGMENT

The authors would like to thank Paul Reeser for his helpful discussions. This work was done when the last two authors were with Bell Labs, Lucent Technologies.

REFERENCES

[1] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Survey*, Vol.13, No.2, June 1981, pp. 185-222.

[2] S. Ceri and G. Pelagatti, *Distributed Databases Principles and Systems*, McGraw-Hill Inc., New York (1984).

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Co., Massachusetts (1987).

[4] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Vol. I*, Computer Science Press, Maryland (1988).

[5] A. S. Acampora, "A Multichannel Multihop Local Lightwave Networks," *Proc. of IEEE GLOBECOM'87*, Tokyo, Japan, November 1987, pp. 1459-1467.

[6] A. S. Acampora and M. J. Karol, "An Overview of Lightwave Packet Networks," *IEEE Networks*, Vol.3, No.1, March 1989, pp. 29-41.

[7] M. G. Hluchyj and M. J. Karol, "ShuffleNet: An Application of Generalized Perfect Shuffles to Multihop Lightwave Networks," *Journal of Lightwave Technology*, Vol. 9, No. 10, October 1991, pp. 1386-1397.

[8] C. Berge, *Graphs and Hypergraphs*, North-Holland Publishing Co., Amsterdam, Holland (1973).

[9] S. M. Hedetniemi and S. T. Hedetniemi and A. L. Liestman, "A Survey of Gossiping and Broadcasting in

Communication Networks," *Networks*, Vol. 18, 1988, pp. 319-349.

[10] W. Knodel, "New Gossips and Telephones," *Discrete Maths*. Vol. 13, 1975, pp. 75-86.

[11] V. S. Sunderam and P. M. Winkler, "Fast Information Sharing in a Complete Network," *Discrete Applied Math.*, Vol. 42, 1993, pp.75-86.

[12] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[13] G.-M. Chiu and K.-S. Chen, "Efficient Fault-Tolerant Multicast Scheme for Hypercube Multicomputers," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 9, No. 10, Oct. 1998, pp. 952-962.

[14] C.-N. Lai, G.-H. Chen and D.-R. Duh, "Constructing One-to-Many Disjoint Paths in Folded Hypercubes," *IEEE Trans. on Computers*, Vol. 51, No. 1, January 2002, pp. 33-45.

[15] L.-J. Fan, C.-B. Yang and S.-H. Shiau, "Routing Algorithm on the Bus-Based Hypercube Network," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 16, No. 4, April 2005, pp. 335-348.

[16] S. Golomb, *Shift Register Sequences*, Holden-Day, 1967.