Fault Tolerant Data Structures

Yonatan Aumann*

Michael A. Bender†

Abstract

We consider the tolerance of data structures to memory faults. We observe that many pointer-based data structures (e.g. linked lists, trees, etc.) are highly nonresilient to faults. A single fault in a linked list or tree may result in the loss of the entire set of data. In this paper we present a formal framework for studying the fault tolerance properties of pointer-based data structures, and we provide fault tolerant versions of the stack, the linked list, and the dictionary tree.

1 Introduction

Motivation. We consider the tolerance of pointer-based data structures to memory failures (e.g. disk sector failures, main memory erasures, accidental overwrites, etc.). We observe that many commonly-used data structures are highly nonresilient to faults. Consider, for example, the linked list. Losing a *single* pointer makes the *entire* tail of the list unreachable. Thus, one fault may result in losing an unbounded amount of data (on average $n/2$ nodes are lost, for a list of size n). Trees, stacks, and other common data structures exhibit similar fragility. Clearly, it is best if faults are avoided entirely. However, faults do occur (if rarely) and it is important to minimize their destructive effect. Loosing an unbounded amount of data as a result of one or two faults seems unacceptable. A naive solution involves replication. If each data item is replicated d times, the data structure becomes resilient to $d-1$ faults. This solution, however, entails a high price both in space and in time: the data structure occupies a factor of d more memory space and requires a factor of d more work for insertions and deletions. Thus, for many applications, replication is impractical.

In this paper we show how to achieve fault tolerance with a significantly lower overhead. We introduce new fault tolerant versions of several common data structures. These new versions are highly resilient to faults, but low in time and space overhead.

Fault tolerance. First, we must establish a proper definition of fault tolerance. Consider the linked list example and suppose that we expect at most d faults. If fault tolerance is interpreted to mean loosing no data, then d different locations in storage must be modified during each update. For many application, this is a prohibitively-high overhead. (The space overhead can be reduced using error correcting codes, but at an extra expense in the time complexity). However, we suggest that loosing *no* data is not necessarily the most useful or natural definition of fault tolerance in the context of data structures. It may be acceptable to lose *some* data; the question is how much. Thus, we define a *parameterized* notion of fault tolerance that measures the *amount* of lost data. Clearly, the more faults occur, the more data is lost. Thus, we define the fault tolerance of a data structure to be the *ratio*, given worst-case faults, of the total amount of data lost to the actual amount of data corrupted by the faults. In a data structure with high resilience to faults, this ratio is *small* and *independent of* n (the size of the data structure). Thus, the weakness of the linked list is reflected in the large fault tolerance ratio of $O(n)$. Once a data structure is discovered to contain faults, it should be reconstructed quickly. We require the time complexity of reconstruction to be a small function of f (the number of faults) and *independent* of n.

The methods introduced here can be combined with standard replication techniques to produce a two-tiered fault tolerance structure. Suppose that under normal circumstances we expect at most d_1 faults, but in extremely unlikely circumstances up to $d_2 \gg d_1$ faults may appear. Then, d_1-fold replication, combined with the methods introduced in this paper, provides the appropriate two-tiered resilience: (i) the replication ensures that under normal circumstances *no* data is lost, but entails a d_1 factor overhead, (ii) our fault tolerance methods guarantee that in the rare event of more than d_1 faults, little data is lost.

Connectivity. Existing data structures are fault prone because of their low connectivity. For example, the graph of a

*Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel. Email: aumann@cs.biu.ac.il.

†Aiken Computation Laboratory, Harvard University, 33 Oxford Street, Cambridge, MA 02138. Email: bender@das.harvard.edu.

linked list is only 1-connected. To obtain higher fault tolerance we must use graphs with higher connectivity. As static structures such graphs are well known. However, here we need graphs that enable *dynamic* insertion and deletion of nodes, while constantly maintaining the high connectivity of the graph. In addition, the topology of the graph must correspond to that of the original data structure, so that the operations of the original data structure can be carried out. Finally, the requirement that the reconstruction time be independent of the size of the data structure means that the high connectivity of the graph must be a local feature (e.g. cannot be maintained by the use of long paths).

Our Results. In Section 2 we provide a formal framework for studying the fault tolerance of data structures. We formally define the degree of fault tolerance of a data structure and show how a fault tolerant data structure can *emulate* a fault prone one. Using these definitions, we provide basic fault tolerant versions of the stack, the linked list, and the binary search tree. We describe the following in Sections 3-6:

- A fault tolerant stack, providing resilience for up to d faults (d is a parameter). The fault-tolerant version occupies only an $O(1)$ factor more space, and operations (push and pop) take $O(1)$ time. With f faults, $O(f \log f)$ nodes are lost (independent of n).
- A fault tolerant linked list, resilient up to d faults (d is a parameter). The space and space overhead is $O(1)$ (time overhead amortized). With f faults, $O(f \log f)$ nodes are lost.
- A fault tolerant binary search tree, resilient up to d faults (d is a parameter). The space and time overhead is $O(1)$ (time amortized). With f faults, $O(f \log f)$ nodes are lost. The solution assumes that nodes of the original tree occupy $\Omega(d)$ space.
- Expander-based versions of the above (stacks, lists, and trees). All guarantee $O(f)$ lost nodes for f faults.

The reconstruction time is a small polynomial in f and d (independent of n). The node degree is constant.

Applications. One of the major goals in file system design is quick recovery from an inconsistent state in the system's metadata (the data structures of a file system). The metadata may reach an inconsistent state, for example, as a result of power failures, undetected disk errors, or internal bugs in the file system. Inconsistent metadata often causes the system to crash. In today's systems, the computer may be unusable for well over an hour during reconstruction (e.g., accomplished using the function `fsck` in UNIX).

We suggest that faster recovery may be achieved by replacing the the current data structures with more fault tolerant ones, in this spirit of this paper. Note that because we are concerned with fast recovery, the lost data do not have to be inaccessible forever, but only until a more thorough recovery is completed. Meanwhile, the system is operational.

In many applications, continuous functionality of the system is a prime concern (e.g. airline reservation systems). Such applications have hardware solutions that provide full fault tolerance (e.g. lots of redundant hardware). Unfortunately, these solutions are often very expensive and restricted to mission-critical applications. Software-based fault tolerance in the spirit of this paper, provides an alternative cost-effective solution for the less-demanding applications. It allows for quick recovery of the system as a whole, while the limited amount of lost data can be recovered in the background, using more lengthly procedures.

Related Work. Computing in the presence of memory faults is studied in many contexts. Rabin [22] presents a fingerprinting scheme via random polynomials for recognizing errors in memory. There is a large body of literature on error-correcting codes, useful for memory or transmission fault tolerance [2, 20]. For example, certain Reed-Solomon codes are currently used in compact discs. Rabin [23] introduces the Information Dispersal Algorithm, which has applications to efficient fault tolerant disk storage and fault tolerant routing. This algorithm breaks a large file into redundant small pieces in an asymptotically space-efficient manner. Computing systems use redundant arrays of inexpensive disks (RAIDs) to protect against failures in storage [21]. Extra "check" disks store redundant information so that when a disk fails, its data can be reconstructed.

For the data structures of file systems, replication is a central tool to obtain fault tolerance. For example, in the Cedar Filesystem, described by Hagmann in [10], all metadata appears twice on the disk. The filesystem is constructed under the assumption that at most one disk sector or two contiguous sectors fail simultaneously.

Once a crash occurs in the file system, it is important to recover as quickly as possible. To aid the recovery, some updates are stored sequentially in a buffer called a log. The idea of logging, originally from database systems [9], is currently used in some file systems for fast recovery [10, 15, 25, 26].

Fault tolerance with respect to processor failures is widely studied, but is out of the scope of this paper (see [14, 16, 1, 7, 3, 8]). The fault tolerance of networks, including many specific architectures (e.g. the mesh, hypercube), was studied both with respect to routing and with respect to computing (see [11, 12, 13, 24, 5, 6]). A network architecture especially designed for fault tolerance is described in [19].

Kuttan and Peleg [18, 17] consider the problem of correcting an illegal state of a distributed network, and define the notion of *fault local mending*. A distributed correction algorithm is a *fault local mending* if its time complexity depends only on the number of failed nodes, not on the size of the entire network. In the discussion in [17] the authors rise the possibility of extending their notion to sequential data structures, introducing the notion of fault local mendable data structures. This notion is related, but distinct from the one introduced in this paper.

2 Terminology: Faults, Reconstruction, and Emulation

Data Structures and Data Structure Schemes. A data structure is characterized by the operations it supports *and* by their implementations. Thus, we define a *data structure scheme* $\mathcal{S}$ to be a set of procedures $\{P_1^{\mathcal{S}}, \ldots, P_n^{\mathcal{S}}\}$ that create and manipulate a data structure. An *instance* of data structure $\mathcal{S}$ is any specific structure resulting from the application of a sequence of the procedures of $\mathcal{S}$. For pointer-based data structures, we view instances as directed graphs, with the data fields in the nodes and the edges representing pointers.

The nodes of a data structure graph are only accessed via pointers. Most of these pointers are themselves located in other nodes of the graph. However, the data structure must also allow access to the structure from the "outside world". In the linked list, for example, the pointer to the head of the list is in a fixed location and can be readily accessed. The queue has two such pointers: one to the front of the queue and one to the end. We call the set of fixed pointers to a structure the *handle* of the structure. Formally, the handle $H(\mathcal{S})$ of data structure $\mathcal{S}$ is a set of labeled nodes $\{h_1, \ldots, h_k\}$. Each *handle node* h_i stores a pointer that can point to any node of the data structure. In addition, the handle node may store auxiliary information regarding the pointer or the data structure.

Faults and Reconstruction. We assume that when a node becomes faulty all the data contained in the node and all the outgoing pointers are lost. Faults are detectable upon access. Trying to access a faulty node results in an error message.

We are interested in reconstructing the data structure after some faults are detected. However, full reconstruction may be impossible. For example, the data structure graph may become disconnected and some nodes inaccessible. Thus, we want to reconstruct the data structure on the subset of the remaining nodes. Since some information is necessarily lost, we must define what we mean by reconstruction. Intuitively, we require that: (i) the reconstruction includes as many nodes as possible, and (ii) the reconstruction maintains the essential topological order among the remaining nodes. For example, in the linked list we should maintain the "before-after" order. Thus, we define reconstruction *with respect to a set of relations* R induced by the topology of the graph.

Definition 1 *Let $\mathcal{S}$ be a data structure scheme, and let $R = \{R_1, \ldots, R_t\}$ be a set of relations induced by the graph structure of (instances of) $\mathcal{S}$. Let $S = (V, E)$ be an instance of $\mathcal{S}$. We say that $S' = (V', E')$ is a* reconstruction of S with respect to R *if*

1. *the graph S' is a valid instance of $\mathcal{S}$.*
2. *for each $R_i \in R$, if R_i is a k-relation, then for each k-tuple $\vec{v} \in {V'}^k$, $\vec{v}$ has the relation R_i in S iff it has the relation R_i also in S.*

We call nodes that appear in S but not in S' *lost* nodes. Note that a node may be lost even if it is not faulty. We seek data structures for which all instances can be efficiently reconstructed with a minimal loss of nonfaulty nodes. However, if the number of faults is greater than the size of the handle, then the entire graph may be inaccessible. Thus, for any data structure $\mathcal{S}$, the size of the handle is an upper bound on the number of faults $\mathcal{S}$ can withstand.

Definition 2 *Let $\mathcal{S}$ be a data structure scheme and R a set of relations. We say that $\mathcal{S}$ is* p-fault tolerant with respect to R for up to d faults *(p is a function), if there exists an algorithm A satisfying the following. For any instance S of $\mathcal{S}$, if there are $f \leq d$ faults in S, then algorithm A, on input of the faulty S, outputs a reconstruction S' of S with respect to R, such that the number of lost nodes is bounded by $p(f)$. The running time of A must be polynomial in f and d.*

We seek data structures for which the function p and the running time of A are slowly growing functions and independent of n (the total size of the instance).

Emulation. Most common data structures do not lend themselves to efficient or useful reconstructions. Thus, we introduce fault-tolerant versions of some common data structures. The new versions *emulate* the behavior of the original ones while supplying a higher degree of fault tolerance.

Definition 3 *Let $\mathcal{S} = \{P_1, \ldots, P_k\}$ be a data structure scheme (P_i's are procedures). Data structure scheme $\bar{\mathcal{S}} = \{\bar{P}_1, \ldots, \bar{P}_k\}$ is an* emulation *of $\mathcal{S}$ if*

1. *For each i, $\bar{P}_i$ has the same user interface as P_i.*
2. *For any sequence of invocations of procedures from $\mathcal{S}$, invoking the corresponding procedures from $\bar{\mathcal{S}}$ results in the same output to the user.*

Three measures characterize the quality of the emulation: time, space, and node degree.

Definition 4 *Let $\bar{\mathcal{S}}$ be an emulation of $\mathcal{S}$. We say that $\bar{\mathcal{S}}$ is an (α, β, γ)-emulation if*

- *Time: for each procedure $P_i \in \mathcal{S}$ and $\bar{P}_i \in \bar{\mathcal{S}}$ the (amortized) execution time of $\bar{P}_i$ is at most α times that of P_i.*
- *Space: for any S instance of $\mathcal{S}$, the corresponding instance $\bar{S}$ of $\bar{\mathcal{S}}$ occupies at most β times more space.*
- *Degree: for any S instance of $\mathcal{S}$, and corresponding instance $\bar{S}$ of $\bar{\mathcal{S}}$, the ratio between the maximum out-degree of nodes in S to that in S is at most γ.*

We say that $\bar{S}$ is a constant *emulation, if it is an $(O(1), O(1), O(1))$ emulation.*

3 Fault Tolerant Stacks

The *stack* data structure supports two operations: `Push`(x) and `Pop`. An instance graph takes the form of a directed path, with the handle *Top* pointing to the first node in path and the last node pointing to Null. Each node x has one data field *x.value* and one outgoing pointer *x.next*. The essential topological order of the stack is the "above-under" relation among nodes. The stack data structure is highly non fault tolerant. We describe a new family of stack-like data structures, d-FTstack, $d = 1, 2, \ldots$, such that the d-FTstack is $O(f \log f)$-fault tolerant for up to d faults, with respect to the "above-under" relation.

We describe the graph structure of (instances of) the FT-stack. The d-FTstack graph is composed of a sequence of *layers*, $L_{\lceil (n/2d)-1 \rceil}, \ldots, L_0$, where layer L_0 is on the bottom of the stack and $L_{\lceil (n/2d)-1 \rceil}$ is on the top. Each layer L_i (except possibly for the top one) consists of $2d$ nodes $L_i = \{x_{i,0}, \ldots x_{i,d-1}\}$. Every $\log d + 1$ layers constitute a butterfly structure. A sample graph appears in Figure 1. Each node $x_{i,j}$ has two outgoing edges: *straight* and *diagonal*. The *straight* edge points from $x_{i,j}$ to $x_{i-1,j}$, if $i \neq 0$ and to Null otherwise. Let $j^{(i)}$ be the integer whose binary representation is identical to that of j except for the (i mod $2d$)-th bit, which is flipped (e.g. for $d = 2$, $3^{(22)} = 1$). The *diagonal* edge points from $x_{i,j}$ to $x_{i-1,j^{(i)}}$, if $i \neq 0$ and to Null otherwise. The lexicographic order of nodes corresponds to the order in the stack. The top layer may be incomplete. The handle to the d-FTstack consists of $2d + 1$ nodes. One node, *Top*, points to the current top of the stack. The additional $2d$ handle nodes point to the top $2d$ nodes of the stack. The FTstack procedures `Push()` and `Pop` add and remove the nodes in lexicographic order and maintain the pointer structure.

Claim 1 *For any d, the d-FTstack is a constant emulation of the stack.*

Faults and Reconstruction. When a fault is detected, reconstruction is performed. The reconstruction procedure operates in two phases:

Pop phase: Pop nodes from the FTstack and place them in auxiliary storage. Try to reach each node via both of its incoming pointers. If both are unavailable or if the node is faulty, discard the node. The Pop phase ends when $2d$ consecutive reachable nodes are encountered or when the bottom of the FTstack is reached. At this point the remaining FTstack is functional and has no apparent faults. (Additional faults may still exist further down the FTstack. Run the reconstruction procedure when these are encountered.)

Reinsert phase: Reinsert the nodes using the FTstack `Push()` procedure.

The reconstruction maintains the order of the nodes in the original FTstack.

Claim 2 *If there are f faults then at most $O(f \log f)$ nodes are lost.*

Proof: Let F be the set of faulty nodes. W.l.o.g. assume $f = |F|$ is a power of 2. We say that a node is *inaccessible* if it is lost but not faulty. Consider an inaccessible node x_0 of level L_i. First, suppose level $L_{i+\log f+1}$ exists. Let $A(x_0)$ be the set of nodes of $L_{i+\log f+1}$ that are ancestors of x_0. Since $f < 2d$, we have $|A(x_0)| = 2f$. The straight edges of the butterfly provide node-disjoint paths from the handle to each node in $A(x_0)$. Thus, at most f nodes of $A(x_0)$ are lost. Let $B(x_0)$ be the accessible, nonfaulty nodes of $A(x_0)$. Thus, $|B(x_0)| \geq f$. Let $T(x_0)$ be the graph of the nodes linking $B(x_0)$ and x_0. The nodes of $T(x_0)$ constitute a binary tree. For node $z \in T(x_0)$, let $d(z, x_0)$ be the distance from z to x_0. Let

$$w(z, x_0) = \begin{cases} 2^{-d(z,x_0)+1} & z \text{ is in } T(x), \\ 0 & \text{otherwise.} \end{cases}$$

Since x_0 is inaccessible, for each $y \in B(x_0)$ there must be a faulty node $z \in T(x_0)$ along the path from y to x_0. A node z can lie on at most $w(z, x_0) \cdot f$ such paths. Thus, for x_0 to be inaccessible, it must be that $\sum_{z \in F} w(z, x_0) f \geq f$. Thus, $\sum_{z \in F} w(z, x_0) \geq 1$.

For the case where $L_{\log f+1}$ does not exist (i.e. x_0 is close to the top) a similar argument also gives that $\sum_{z \in F} w(z, x_0) \geq 1$.

Let I be the set of inaccessible nodes. Consider a faulty node z. Node z is in distance 1 from two nodes, distance 2 from four nodes, and so on. Thus, summing over all trees $T(x)$ that contain z, we obtain

$$\sum_{x \in I} w(z, x) \leq \sum_{i=1}^{\log f} 2^i 2^{-i+1} = 2 \log f.$$

Thus,

$$|I| \leq \sum_{x \in I} \left(\sum_{z \in F} w(z, x) \right) =$$

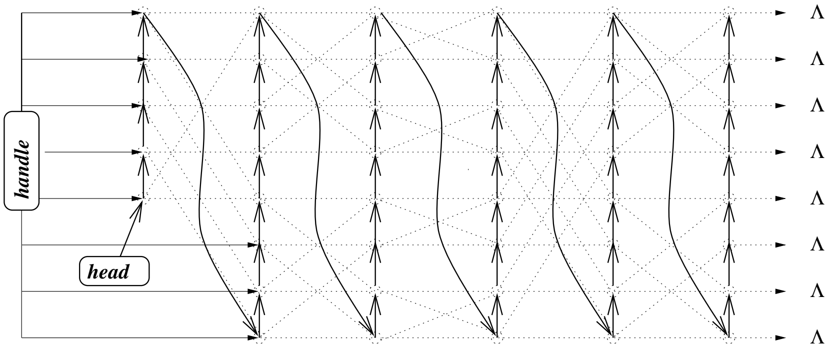

Figure 1. A d-FTstack, where $d = 4$ and $n = 45$.

$$= \sum_{z \in F} \left(\sum_{x \in I} w(z, x) \right) \leq 2f \log f.$$

Thus, the total number of lost nodes is $|I| + |F| \leq 2f \log f + f$. ■

Claim 3 *For any f and d, the d-FTstack reconstruction procedure completes in $O(fd)$ steps.*

Proof: Popping and reinserting a node takes a constant number of steps. The Pop phase completes once $2d$ consecutive accessible nodes are encountered. We prove that with f faults, at most f consecutive layers have inaccessible nodes. Since each layer consists of $2d$ nodes, the total work is $O(fd)$.

The proof is by induction on f. For $f = 1$ the claim holds. Assume the claim holds for f faults. By contradiction assume that there exists a placement of $f + 1$ faults such that k consecutive layers $L_{i+k-1}, \ldots, L_i$ all contain inaccessible nodes, and $k > f + 1$. W.l.o.g. assume $i = 0$ and $k = f + 2$. Denote by F_i and U_i the set of faults and the set of inaccessible nodes of layer L_i, receptively. Let j_0 be the smallest index such that $|F_{j_0}| \neq \emptyset$. Let y be any node in F_{j_0}. Consider the situation if y were not faulty. For all i, let U'_i be the set of inaccessible nodes in L_i if y were not faulty. By the inductive hypothesis if y were not faulty then at most f consecutive layers have inaccessible nodes. Thus, there must be a j_1, with $j_1 \geq f + 1 > 0$, such that $U'_{j_1} = \emptyset$ (but $U_{j_1} \neq \emptyset$). Clearly, $j_1 \leq j_0$. Thus, $j_1 - 1 < j_0$ and $f_{j_1-1} = 0$. Thus, there are no faults in L_{j_1-1} and $|U'_{j_1-1}| = |U'_{j_1}| = 0$. Consider $x \in U_{j_1-1}$. Denote by x_1 and x_2 the two predecessor nodes of x in L_{j_1}. Since x is inaccessible it must be that both x_1 and x_2 are inaccessible. However, both are accessible if y were not faulty. Thus, both are descendants of y. no faults, and all nodes are accessible). Thus, there are *two* directed paths from x to y. Clearly, the distance from y to x cannot be more is at most $\log 2d$ (otherwise there are all $\log 2d$ layers with However, in the graph of a d-FTstack the descendents of any node form a tree for the first $\log d$ layers. In contradiction. ■

Theorem 1 *For any d, the d-FTstack is $O(f \log f)$-fault tolerant for up to d faults, and is a constant emulation of the stack.*

4 Fault Tolerant Linked Lists

The linked list supports the of following operations:

- `Insert`(v, p): inserts v before node p (where p is a pointer).
- `Remove`(p): removes node p.
- `Get`(p): returns value of node p.
- `Next`(p): returns (pointer to) the node following p.
- `Head`: returns (pointer to) first node in the list.

The essential relation between nodes is the 2-relation "before-after". The linked list is highly non fault tolerant. We introduce a family of fault tolerant data structures, d-FTlist, $d = 1, 2, \ldots$ that emulate the linked list. For each d, the d-FTlist is $O(f \log f)$-fault tolerant for up to d faults and is a constant emulation of the linked list.

Instance graphs of the linked list are directed paths. Thus, the stack and the linked list have the same graph structure. In the previous section, we used a layered graph structure to obtain a fault tolerant emulation of the stack. We use a similar structure for the fault tolerant linked list. However, the essential difference between the stack and the linked list is that nodes in the linked list are inserted and deleted anywhere in the graph, whereas nodes in a stack are pushed and popped from the top. Thus, the main difficulty in constructing a fault tolerant list is maintaining the structure throughout the dynamic changes. A sample graph of an FTlist appears in Figure 2.

The Skeleton. Assume $2d = 2^{\ell}$. The graph of the d-FTlist is composed of a sequence of *blocks* $B^0, \ldots, B^{i_{head}}$.

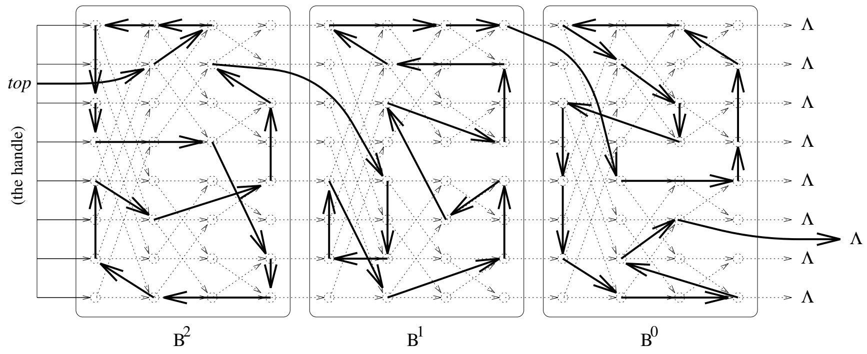

Figure 2. A d-FTlist, where $d = 4$.

Each block B^i consists of $2d \log(2d)$ *vertices* arranged in a butterfly structure. In addition, B^i has one special *header node*, *header*(B^i). Vertices in the last level of block B^i point to those in the first level of B^{i-1}. The handle of the d-FTlist consists of $2d + 1$ nodes. One handle node, *head*, points to the head of the list. The remaining handle nodes, $h[k]$, $k = 0, \ldots, 2d - 1$, point to the nodes in the first level of the first block. We call this block and vertex structure the *skeleton* of the graph.

Folding. The original linked list is *folded* onto the skeleton as follows. Each node x of the linked list is mapped to a vertex s of the skeleton. Vertex s stores the entire information in x, including the data field and the *next* pointer. At most one list node is mapped to any skeleton vertex. The mapping maintains the order of nodes across blocks. Within each block, the nodes are mapped arbitrarily. Specifically, if x and y are nodes of the original linked list and x is before y, then either x and y are in the same block, or x is in a block before y. The empty vertices of each block B^i are chained in a list *free*$_i$. The head of *free*$_i$ is stored in *header*(i). In addition, *header*(i) maintains the variable Load_i, which records the total number of nodes mapped onto vertices of B^i. Let $M = 2d \log(2d)$ be the number of vertices in a block. We maintain the following invariant.

Invariant 1 *At all times at least $M/4$ nodes are mapped to each block B^i ($i > 1$), and at most M are mapped to each block B^i ($i \geq 0$). (Only the last block B^0 may contain less than $M/4$ nodes.)*

Procedures. A full copy of the linked list is embedded in the d-FTlist. Thus, implementing the procedures `Head`, `Next()`, and `Get()` is easy with the following small addition. A regular linked list is accessed via a pointer p, which points to (one of) the current location(s) in the list. In the FTlist one pointer is never enough, because any single node may become faulty. Thus, in addition to the pointer p, the user holds a set of $2d$ pointers to the nodes of first level of the current block. The pointers are initialized in `Head`, and updated as the current location changes from one block to the next. The amortized cost per execution of `Next()` is $O(1)$.

`Insert`(v, x) involves the following steps:

1. Insert the value v before x in the embedded copy of the linked list. Let y be the new node.
2. Suppose x is in block B^i. Map the new node y onto a free vertex s in B^i.
3. Update the list *free*$_i$, and advance Load_i by 1.
4. If $\text{Load}_i = M$ then *split* B^i into two blocks.

To split a block, first create two skeleton blocks. Then, insert the first half of the nodes in one block and the second half in the next. We omit the details. Splitting completes in $O(M)$ operations.

`Remove`(x) is the reverse procedure:

1. Delete x from the embedded copy of the linked list.
2. Suppose x is mapped to vertex $s \in B^i$. Delete x from s, and add s to *free*$_i$. Decrease Load_i by 1.
3. If $\text{Load}_i < M/4$ (and B^{i-1} exists) then execute one of the following: If $\text{Load}_{i-1} \leq 3M/4$ then *join* B^i with B^{i-1}. Otherwise, *divide* the nodes evenly between B^{i-1} and B^i.

Joining/dividing is completed in $O(M)$ operations.

Tags. Within each block, the nodes can be mapped on the skeleton in an arbitrary order. Hence, we provide an additional facility so that the order of nodes in the same block can be reconstructed in the case of faults. We have each node x of the linked list store a *tag*, *tag*(x), which is an integer of size $O(\log n)$. Dietz and Sleator [4] show how to maintain such tags, such that for any two nodes x and y, *tag*$(x) <$ *tag*(y) iff x is in front of y. Thus, the order between any two nodes can be recognized in constant time. The [4] algorithm has an $0(1)$ amortized work per insert/delete. We note that since there are n nodes in the list, the size of the tag is that of a pointer.

Reconstruction. When a fault is detected, reconstruction is performed as follows (recall that at all times the user of the structure holds $2d$ pointers to the first layer of the current block):

1. Starting from the first layer of the block, advance one level at a time, until a block with no faults is encountered. Gather list nodes en route.
2. Using the tags, find order between nodes.
3. Reconstruct blocks for the nodes.

Theorem 2 *For any d, the d-FTlist is a constant emulation of the linked list, and is $O(f \log f)$-fault tolerant with respect to the "before-after" relation, for up to d faults.*

Proof: Each vertex has a constant size and a constant outdegree. At least a quarter of the vertices are not empty. Thus, the d-FTlist takes an $O(1)$ factor more space than the linked list. Functions `Get()`, `Next()`, and `Head` run in a constant number of steps. Regular insertion and deletions require $O(1)$ operations. After every $\Omega(d \log d)$ insertions or deletions in a block, the block must be split or joined. These tasks consist of $O(d \log d)$ operations on the skeleton followed by $O(d \log d)$ insertions. Thus, the amortized work per insertion or deletion is $O(1)$.

By Claim 2 with $f < d$ faults, at most $O(f \log f)$ vertices of the skeleton are lost. Each vertex stores at most one list node. Thus, at most $(f \log f)$ list nodes are lost. Furthermore, at most $f + 1$ consecutive blocks have inaccessible nodes. Thus, the reconstruction completes in $O(fd \log^2 d)$ operations. ∎

5 Fault Tolerant Trees

We consider a binary search tree that supports the following procedures:

- `Insert`(v) insert node v. A node is inserted as a new leaf or between a parent and child.
- `Remove`(v): remove v from the tree. Only nodes having one or zero children can be deleted.
- `Find`(v): find key v.

We assume that all operations on the tree start at the root, which is the handle of the structure. We construct a family of data structures d-FTtree, $d = 1, 2, \ldots$ that emulate the dictionary tree and that are fault tolerant with respect to the "above-under" and "left-right" order of leaves. We ensure that the depth of a leaf in the reconstructed structure is no more than in the original tree.

Consider a tree T. As in the FTlist, we use a block structure as a skeleton. Each block of the d-FTtree consists of $M = 2d \log 2d$ vertices interconnected in a *wrapped butterfly* structure. Also as in the FTlist, the blocks have a header storing the free list and the load. The details of these auxiliary structures are omitted. The nodes of T are mapped to vertices of the blocks, so that each vertex holds at most one node. (We use the term *vertex* to refer to the vertices of the blocks, and the term *node* to refer to the nodes of the tree T.) We say that the block *contains* the nodes mapped to it. A sample FTtree is depicted in Figure 3.

The blocks are logically arranged in a tree structure, which we call the *block tree* and denote by BT. The mapping of the nodes of T to the blocks maintains the following conditions:

1. Let Child(B) be a child of B in BT. Let $T_1, \ldots, T_k$ be the connected components of the nodes contained in Child(B) (each T_i is a subtree of T). Then, the root of each T_i is a direct child of a node in B.
2. If R-Sib(B) is a right sibling of B, then in T the nodes contained in R-Sib(B) are to the right of those in B.

If the nodes in a block B are all in one connected component, we say that B is a *uni-component* block. Otherwise B is a *multi-component* block.

Blocks are interconnected using *wide links*. To construct a wide link between blocks B and B', we have corresponding vertices in B and B' point to each other. (Recall that all blocks have the same butterfly structure.) Wide links are maintained between the following blocks:

1. If a block B has children in BT, then there is a wide link between B and its leftmost child.
2. If B has siblings, then there is a wide link from B to its sibling immediately to the right.

Thus, all children of a given block are connected in wide-link linked list, rooted in the parent. The handle to the FTtree consists of $2d$ pointers to the first layer of the root block.

We maintain the following invariants:

1. At most M nodes are mapped to each block.
2. Only uni-components blocks have children.
3. If block B has a child then B contains at least $M/4$ nodes.
4. If B contains less than $M/4$ nodes, then either B is the only block in the tree, or its immediate sibling (to the right or the left) is uni-component and contains at least $M/4$ nodes.

Claim 4 *1. Each vertex of the FTtree has outdegree at most* 8.

2. The size of the FTtree is linear in the size of T.

3. Each block has at most $2M$ child blocks.

Proof:

1. The skeleton structure uses six of the outgoing pointers: two pointers for the butterfly structure within a

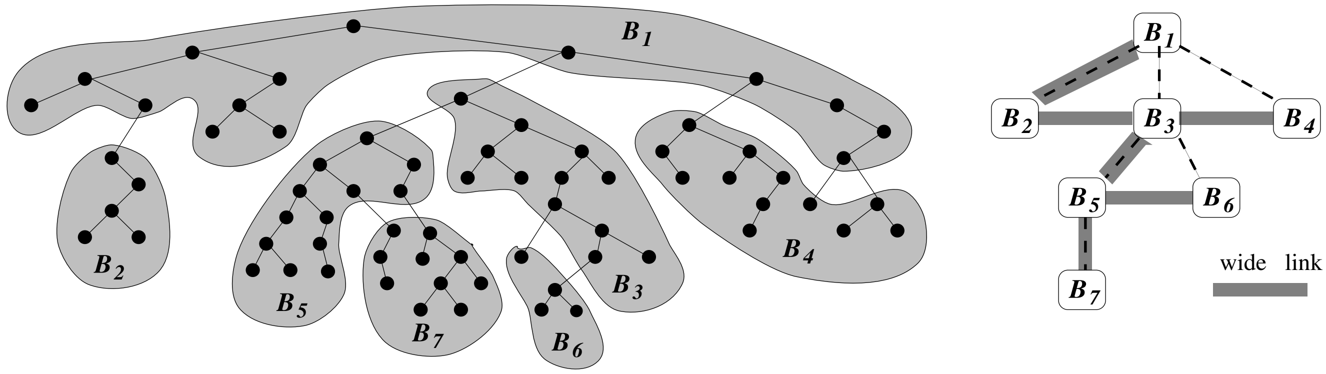

Figure 3. A d-FTtree, with $M = 32$. Left: mapping of nodes to blocks. Right: the *block tree* (logical links in dashed, wide links in grey). Blocks B_1, B_3 and B_5 have children. They are *uni-component* and contain more than $M/4$ nodes. Blocks B_4, B_6, B_7 are *multi-component*. They have no children. Block B_2 contains less than $M/4$ nodes, but has an immediate sibling, B_3, that is uni-component and contains more than $M/4$ nodes.

block, and pointers to the corresponding vertices in the child block, parent block, right-sibling block, and left-sibling block. The remaining two pointers maintain the structure of T mapped onto the vertices of the skeleton; a node in T may have two children.

2. The proof is by accounting. We assign each block B containing less than $M/4$ nodes to a block containing at least $M/4$ nodes as follows. If B is an only child, we assign it to the parent, which by the invariant contains at least $M/4$ nodes. Otherwise, we assign it to its immediate sibling, which by the invariant contains at least $M/4$ nodes. Thus, at most three extra blocks are assigned to each block with $M/4$ nodes.

3. A block contains at most M nodes. Each node has at most two children in T. In the worst case, each child is in a separate child block.

Dynamics. *Insertion:* When a new node is inserted, it is mapped to the block containing the node's parent. If the new block then contains M or more nodes, it is *split* into two blocks. There are two kinds of splits: *horizontal* and *vertical*. A vertical split yields a parent and child; a horizontal split yields two siblings. A uni-component block undergoes a vertical split. A multi-component block first undergoes a horizontal split, and then, if necessary, a vertical one. The split procedure takes $O(M)$ operations. The main concern is to amortize the cost over $O(M)$ inserts/deletes. Thus, the split procedure maintains that if B is a block created during the split then:

1. Block B contains at most $2M/3$ nodes. Thus, there need to be at least $M/3 = O(M)$ new inserts to B before it is split again.

2. Either B contains at least $1/3M$ nodes, or it has a uni-component neighbor with $1/3M$ nodes. Thus, at least $M/12 = O(M)$ nodes must be deleted from B before it must undergo a merge (see below).

Deletion: If the number of nodes in a block falls under $M/4$ and it does not have a uni-component immediate sibling with more than $M/4$ nodes, it is merged with a sibling, a parent, or a child. Again, we guarantee that the resulting blocks do not undergo another join or split for at least $\Omega(M)$ additional inserts and deletes. The join procedure takes $O(M)$ work. Thus, the amortized work per insert/delete is $O(1)$.

Tags. We use tags of size d. Consider a node x mapped to block B. Let T_i be the component of x in B, and let Parent(B) be the parent B. Let r_i be the root of T_i and r_B be the root of the (single component) of Parent(B). The tag of x is

$$tag(x) = < \text{path from } x \text{ to } r_i \text{ , path from } r_i \text{ to } r_B >$$

Reconstruction. Reconstruction follows the tree structure of the blocks. To reconstruct a block, first reconstruct recursively all of its children. The M-wide linked list of child blocks guarantees that *all* child blocks are reachable. After all child blocks have been reconstructed, the block itself is recosntructed. The reconstruction procedure first uses the butterfly links of the skeleton to *find* the nodes. Then it reconstructs the interconnections among the nodes using the tags. In addition, the tags allow the procedure to recreate lost nodes, if they are missing from a path to an existing nodes. A description of the reconstruction procedure appears in Figure 4.

Reconstructing a Tree Block B

1. Reconstruct child blocks { recuse }
2. Gather all accessible nodes of the block
   (using butterfly structure)
3. Set $A :=$ (recovered nodes)
4. Set $T := \{tag(x) | x \in A\}$
5. **foreach** $t = < P, Q > \in T$ **do** { P, Q are paths }
6. **foreach** P' prefix of P
7. **if** $< P', Q > \notin T$ { missing nodes }
8. $T := T \cup \{< P', Q >\}$
9. **if** B has children { B is uni-component }
10. Set $\hat{Q} = \{Q | \forall x \in A, tag(x) = < \cdot, Q >\}$
11. Set $R :=$ (roots of components from all children)
12. **foreach** $x \in R$, $tag(x) = < \emptyset, H >$
13. **foreach** H' proper prefix of H
14. **if** $< H', \hat{Q} > \notin T$ **then** $T := T \cup \{< H', \hat{Q} >\}$
15. T contains tags for all necessary nodes in the block
16. **foreach** $t \in T$ create node
17. Map nodes to vertices; set pointers according to tags
18. For each $x \in A$ copy data to corresponding node
19. Set pointers to roots in children blocks

Figure 4. Reconstructing a Block

Claim 5 *Reconstruction takes $O(poly(fM))$ steps.*

Proof: The skeleton vertices of each block are organized in a butterfly structure. Thus, by Claim 2, if there are f faults in block B, then at most $O(f \log f)$ nodes of B are lost. In particular, if there are no faults in the block, all nodes are accessible. Each block has at most $2M$ children. Thus, the recursive procedure for reconstruction visits at most $O(fM)$ blocks. Reconstruction of a block depends on the block and it's children and can thus be completed in $O(\text{poly}(M))$ steps. ∎

Theorem 3 *For any d, the d-FTtree is a constant emulation of the binary dictionary tree, and is $O(f \log f)$ fault tolerant with respect to the "above-under" and the "left-to-right" relation between nodes, for up to d faults.*

6 Fault Tolerance with Expanders

In the fault tolerant data structure presented so far, f faults result in $O(f \log f)$ lost nodes. Using expanders, the number of lost nodes is reduced to $O(f)$.

We first consider the EFTstack (*Expander Fault Tolerant Stack*). As in the FTstack, in the EFstack the nodes are grouped into layers. Here however, the layers are interconnected using a bounded-degree expander (instead of a level of a butterfly). Specifically, let $G_d = (A, B, E)$, $|A| = |B| = 2d$, be a fixed bounded-degree bipartite expander graph, with expansion rate $\alpha > 1$. The graph interconnecting every two consecutive layers of the d-EFstack is isomorphic to G_d. In addition, each node points to the corresponding node in the following layer. Note that d is fixed for any given d-EFT-Stack, and thus G_d can be hard-wired into the code. Implementations of `Pop`, `Push()`, and reconstruction are similar in the EFstack and the FTstack. We obtain:

Theorem 4 *For any d, the d-EFstack is a constant emulation of the stack and is $O(f)$-fault tolerant with respect to the "above-under" relation, for up to d faults.*

Proof: We show that with f faults, there are at most $O(f)$ inaccessible nodes. Let U_i and F_i be as above. By the expansion property, if $|U_i| \leq d$ then $|U_i| \leq |U_{i+1}/\alpha + |F_i|$. Since $\alpha > 1$ and $\sum |F_i| \leq d$, by induction $|U_i| \leq d$, for all i. Thus, the total number of inaccessible nodes is

$$\sum_i |U_i| \leq \sum_{j=0}^{\infty} \frac{1}{\alpha^j} \sum_i f_i = O(f).$$

∎

Similarly, the d-FTlist and d-FTtree are converted into the d-EFTlist and d-EFTtree. The blocks of the d-EFTlist and the d-EFTtree contain $2d$ vertices (instead of $2d \log 2d$, since there is no need for levels within blocks). Two blocks are interconnected by a fixed expander graph. Mapping within each block is in an arbitrary order. The tagging scheme is unchanged. We obtain:

Theorem 5 *For any d, the d-EFTlist is a constant emulation of the linked list and is $O(f)$-fault tolerant for up to d faults.*

Theorem 6 *For any d, the d-EFTtree is a constant emulation of the binary search tree. It is $O(f)$-fault tolerant for up to d faults, with respect to the "above-under" and "left-to-right" relations.*

Acknowledgments.

We are grateful to Pino Italiano for pointing us to reference [4].

References

[1] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 1–10, 1988.

[2] E. R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, 1968.

[3] B. Chor, M. Merrit, and D. Shmoys. Simple constant time consensus protocols in realistic failure models. In *Proceedings of the 4th Annual ACM Symposium on the Principles of Distributed Computing*, pages 152–162, 1985.

[4] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 365–372, 1987.

[5] D. Dolev, J. Halpern, B. Simons, and H. Strong. A new look at fault-tolerant network routing. *Information and Computation*, 72(3):180–196, March 1987.

[6] C. Dwork, D. Peleg, N. Pippenger, and E. Upfal. Fault tolerance in networks of bounded degree. *SiComp*, 1989.

[7] P. Feldman and S. Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 148–161, 1988.

[8] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed commit with one faulty process. *Journal of ACM*, 32(2):374–382, April 1985.

[9] J. Gray. *Notes on Data Base Operating Systems*, pages 393–481. Springer-Verlag, Berlin, 1979.

[10] R. Hagmann. Reimplementing the Cedar File System using logging and group commit. In *11th SOSP*, pages 155–162, December 1987.

[11] J. Hastad, T. Leighton, and M. Newman. Reconfiguring a hypercube in the presence of faults. In *Proceedings of the 28th Annual Symposium on the Foundations of Computer Science*, pages 274–284. IEEE, 1987.

[12] J. Hastad, T. Leighton, and M. Newman. Fast computation using faulty hypercubes. In *Proceedings of the 30th Annual Symposium on the Foundations of Computer Science*, pages 251–263. IEEE, 1989.

[13] C. Kaklamanis, A. Karlin, F. Leighton, V. Milenkovoc, P. Raghavan, S. Roa, C. Thomborson, and A. Tsantilas. Asymptotically tight bounds for computing with faulty arrays of processors. In *Proceedings of the 31st Annual Symposium on the Foundations of Computer Science*, pages 285–296, 1990.

[14] P. Kanellakis and A. Shvartsman. Efficient parallel algorithms can be made robust. In *Proceedings of the 8th Annual ACM Symposium on the Principles of Distributed Computing*, pages 211–221, 1989.

[15] M. L. Kazar, B. L. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S. T. Tu, and E. R. Zayas. Decorum file system architectural overview. In *USENIX*, pages 151–164, Summer 1990.

[16] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 381–390, May 1991.

[17] S. Kutten and D. Peleg. Fault-local mending. In *Proceedings of the 14th Annual ACM Symposium on the Principles of Distributed Computing*, pages 20–27, 1995.

[18] S. Kutten and D. Peleg. Tight fault-locality. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, 1995.

[19] T. Leighton and B. Maggs. Expanders might be practical: Fast algorithms for routing around faults in the multibutterflies. In *Proceedings of the 30th Annual Symposium on the Foundations of Computer Science*, pages 384–389. IEEE, October 1989.

[20] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. Elsevier Science Publishers, Amsterdam, The Netherlands, 1977.

[21] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *11th SOSP*, pages 386–393, 1988.

[22] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR–15–81, Center for Research in Computing Technology, Harvard University, 1981.

[23] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the Association for Computing Machinery*, 36(2):335–348, April 1989.

[24] P. Raghavan. Robust algorithms for packet routing in the mesh. In *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures*, June 1989.

[25] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[26] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *USENIX*, Winter 1993.