Distributed Paging for General Networks

Baruch Awerbuch* Yair Bartal† Amos Fiat†

Abstract

Distributed paging [BFR92, ABF93b, AK95] deals with the dynamic allocation of copies of files in a distributed network as to minimize the total communication cost over a sequence of read and write requests.

Most previous work deals with the *file allocation* problem [BS89, West91, CLRW93, ABF93a, WY93, Koga93, AK94, LRWY94] where infinite nodal memory capacity is assumed. In contrast the distributed paging problem makes the more realistic assumption that nodal memory capacity is limited. Former work on distributed paging deals with the problem only in the case of a uniform network topology.

This paper gives the first distributed paging algorithm for *general* networks. The algorithm is competitive in storage and communication. The competitive ratios are poly-logarithmic in the total number of network nodes and the diameter of the network.

*Johns Hopkins University and Lab. for Computer Science, MIT. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract 9114440-CCR, DARPA contract N00014-J-92-1799, and a special grant from IBM. E-Mail: baruch@theory.lcs.mit.edu.

†Department of Computer Science, School of Mathematics, Tel-Aviv University, Tel-Aviv 69978, Israel. Supported by a grant from the Israeli Academy of Sciences. E-mail: `yairb@math.tau.ac.il`, `fiat@math.tau.ac.il`

1 Introduction

The problem. Many modern information services know no national boundaries or geographical restrictions. This was true of airline reservation services three decades ago, to data processing services of multinational corporations two decades ago, and is now true of an ever increasing mass of data services. The widespread use of the Internet and Internet-related applications such as the World Wide Web is growing fantastically on an annual basis.

With the appearance of the massively parallel machines in the 1980s, it was natural to extend the virtual memory concept from the traditional uniprocessor to distributed shared-memory environment. In other words, the programmer can use the convenient Parallel Random Access Machine (PRAM) abstraction to write the program, which will be then compiled automatically to run on message-passing network-based distributed memory machines.

The *distributed paging* problem deals with the dynamic reallocation of file copies by replicating and migrating files between processors, over a sequence of read and write requests at the processors. Our goal is to minimize the communication cost and memory cost of emulating shared memory, by properly exploiting locality (details in Section 2).

Contribution of this paper. This paper is the *first analytic* treatment of the problem of efficiently compiling shared-memory programs in its full generality, that is universally applicable to *any* computing environment (sequential, parallel, distributed) and *any* input request pattern. It applies to a Sun workstation in your office, personal wireless organizer (Newton), distributed file services such as World-Wide Web, parallel multi-processors such the Connection Machine, etc.

The contribution of this paper is designing *first* competitive centralized and distributed solution for *general networks*, with poly-logarithmic overheads in space and communication.

Existing work. Certainly, locality-exploiting data management has been actively discussed in the context of parallel architectures [ACJ$^+$91, D$^+$89, John92] and compilers [LW91, LW92, Coop92], as well in more general context of data organization, say, in a distributed object store [Lisk92, RD90, Stam84, LMW91, PZ91], distributed databases and services in computer networks. The amount of work on the subject in the systems community is overwhelming. For example, the 1981 survey paper by Dowdy and Foster [DF82], and 1990 survey paper by Gavish and Sheng [GS90] dealing with the file allocation problem, cite close to a hundred references.

In theory community, only special restricted versions of the problem have been previously addressed, such as

- *Uniprocessor paging* [ST85, KMRS88, FKL+88, RS89]: the underlying communication networks consists of a single link from memory to cache.
- *File allocation* [BS89, BFR92, West91, CLRW93, ABF93a, WY93, Koga93, AK94, LRWY94]: *infinite* nodal capacity is assumed.
- *Uniform network topology* [BFR92, ABF93b, AK95]: network with *clique topology* where distances between any pair of nodes are the same.

2 The Model and the Problem

2.1 Basics

Network Model. The underlying network topology is modeled by a weighted undirected graph $G(V, E, w)$ where processors are represented by vertices, and edge weights represent transmission costs between corresponding pairs of vertices. We denote by δ its weighted diameter (where edges weight is at least 1) and by $n = |V|$ the number of nodes. The weighted graph need not obey the triangle inequality, but a natural metric space can be defined where the points are processors and the distance between two points u and v, is equal to the length of the shortest path between the processors in the weighted graph, denoted $d(u, v)$.

Each node v has memory size k_v files; it can keep in its memory any subset of size k_v out of the aggregate virtual memory pages $\mathcal{F}$. The total size of aggregate virtual memory is $m = \sum_{v \in V} k_v$.

The Distributed Paging Problem. A distributed paging algorithm receives a sequence of read and write events occurring at different network nodes for some set of files, $\mathcal{F}$. The algorithm produces a sequence of *responses*, each response changing the *configuration*, namely the set of files maintained by each node.

The response is based on the information available to the algorithm. In the case of *off-line* algorithm, this information consists of the past and the future inputs. In the case of *distributed on-line* algorithms, it consists of the past *local* inputs and the part of the current global configuration explicitly transmitted to the location at which the decision is made.

Data is assumed to be organized in indivisible blocks such as files (or pages), consisting of D basic data units. It can be accessed via communication links by paying a charge equal to the data transfer size times the distance traversed (in the above metric space).

More formally, let Q_F be the current set of processors holding a copy of file $F \in \mathcal{F}$.

At any time a processor p may invoke one of two types of requests: *Read* F in p, with an inherent cost of communicating from p to closest replica of F, $d(Q_F, p)$, and *Write* F in p, with an inherent cost of communicating from p to all the replicas of F, that is the weight of a minimum Steiner tree $T(Q_F \cup \{p\})$.

In response the distributed paging algorithm may produce one of two types of operations: *Delete* F from p, at a zero cost, and *Replicate* F into p, at the cost of transmitting a file copy between the closest replica of F and p, $D \cdot d(Q_F, p)$.

2.2 Complexity Measures and Competitive Framework

We will use, as a measure of performance, the "competitive ratio" [ST85, KMRS88] which is defined as the ratio between the costs associated with a on-line algorithm versus the costs expended by an optimal dynamic policy (referred to as "adversary"), who has perfect knowledge of the future, to deal with same sequence of events.

We design competitive distributed algorithms [BFR92, AADW94] for the *distributed paging* problem, i.e. distributed algorithms such that their total communication cost for both data management and control-messages is not much larger than the cost of any adversary.

Our goal is giving an algorithm for distributed paging on arbitrary network topologies. However, similarly to the case of the k-server problem [MMS88] (which can be reduced to distributed paging), the best competitive ratio possible for deterministic algorithms is $\Omega(m)$ [BFR92] (where m is the total size of the aggregate virtual memory). This leads us to using a somewhat different model by defining a weaker but reasonable adversary, for which we can present a distributed paging algorithm with poly-logarithmic competitive ratio.

A similar situation occurs for the uniprocessor paging problem where the cache size, k, is a lower bound on the competitiveness of any deterministic paging algorithm.

The model we adopt here is the one suggested in [ST85, KMRS88, MMS88, RS89, Young91] for uniprocessor paging, where the adversary has a smaller cache size than the on-line algorithm. This can also be viewed as a solution for the problem where the cache size may vary (cf. [Young91]), but the algorithm should pay for enlarging its cache size. This slackness alleviates one of complications in [ABF93b], namely what to do with the last copy of the data being erased.

More precisely, we measure the performance of an on-line distributed paging algorithm by two measures: one is the ratio between its cost for serving the request sequence (including it's communication cost in the distributed setting.) and that of the adversary, and the second is the ratio between its cache size and that of the adversary's in every processor.

Definition. Given an on-line distributed paging algorithm and any adversary, Adv, let k_p denote the on-line memory capacity of processor p, and let h_p denote the adversary's memory capacity for processor p.

Let c and s be constants at least 1. If for any adversary, Adv such that for all p, $k_p \geq s \cdot h_p$, the algorithm is c-competitive against Adv, then we say the algorithm is (c, s)-competitive.

In this terminology, the following theorem was proved in [ST85, KMRS88, RS89]:

Theorem 2.1 *There exist deterministic (2,2)-competitive uniprocessor paging algorithms.*

Thus, in the uniprocessor case the assumption of some small advantage in memory for the on-line algorithm reduces the competitive ratio dramatically. In this paper we show that a similar theorem holds for the distributed paging problem.

2.3 Contributions of This Paper

The major contribution of this paper is the first competitive on-line distributed paging algorithm for arbitrary network topologies. We prove the following theorem:

Theorem 2.2 *There exist deterministic* $(\mathrm{polylog}(n, \delta), \mathrm{polylog}(n, \delta))$*-competitive distributed paging algorithms for general networks.*

The construction of this algorithm is based on several other results, of independent interest:

- A *reduction* theorem from read/write distributed paging problem to read-only distributed paging problem. Showing that copy consistency issues can be ignored in constructing distributed paging algorithms.
- A *read-only* distributed paging strategy based on a hierarchy of uniprocessor paging strategies.
- A file allocation algorithm for *dynamic* networks.
- A theorem concerning a natural combinatorial problem we call *vertex inclusion-exclusion*.

3 Overview of Algorithms and Proofs

The algorithm we give deals with the most general problem of distributed paging, and is constructed in a modular way from a number of conceptually simpler algorithms (See Figure 1). Specifically, the algorithm and it proof are constructed in stages as follows:

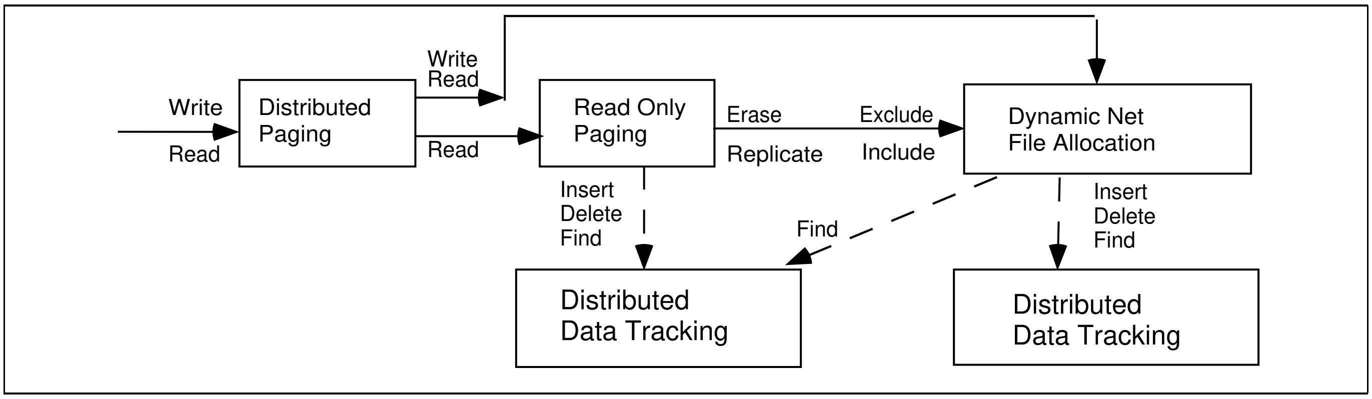

Figure 1: The subroutine structure used in our algorithm.

- In Section 4 we give the read-only distributed paging algorithm Hierarchical Paging. The major idea is to use uni-processor paging on "areas of activity". These areas are defined using the sparse partitions of [AP90]. After sufficiently many accesses to a file have been issued by processors within such an "area of activity", a copy of the file is fetched into this area. For every such "area of activity" we simulate a uni-processor paging process that determines what files are to be ejected/erased to make space for new files.
- Next we show that the problem of read/write distributed paging can be reduced to the problem of read-only distributed paging, which enables to eliminate the problem of "copy consistency" and thus enable to isolate the issue of conflicting demand for shared memory pool. In Section 5 we describe the reduction algorithm, called DP, which makes use of two different algorithms:

  – A file allocation algorithm for dynamic networks, DynDFA, problem (Section 5). This is a version of distributed file allocation where processors may suddenly die out or reawaken after being unavailable. This problem may prove to be of independent interest. Figure 5 describes an instance of the dynamic file allocation problem.

    The proofs concerning our dynamic file allocation algorithm make use of a the "vertex inclusion-exclusion problem" (Section 5.3). Given a sequence of inclusions and exclusions to a subset of graph nodes, each associated with an inherent cost of the minimal distance to a node currently in the set. We show the cost of deletions and the cost of insertions are within a gap of at most a logarithmic factor apart.

  – A read-only distributed paging algorithm. This algorithm is simulated by the read/write distributed paging algorithm, its actions are then used to generate on-line events to the dynamic distributed file allocation algorithm, these include read/write file allocation events as well as include/exclude processor events. The actions of the dynamic file allocation algorithm are then used to determine the true actions of the read/write distributed paging algorithm.

For the distributed implementation of our algorithms we use a distributed data tracking mechanism (Section 4.1) to efficiently locate file copies in the network.

4 Read-Only Distributed Paging for Arbitrary Networks

This section presents the read-only distributed paging algorithm, HIERARCHICAL PAGING (HP).

The algorithm uses the hierarchical graph decomposition of [AP90], and runs in every cluster of the decomposition a simulation of some uniprocessor paging algorithm, U.

For simplicity of description we assume that algorithm U is defined for any cache size, and that the same algorithm is used for all clusters, although these assumptions are not important.

We prove that if U is (γ, τ)-competitive for uniprocessor paging then the HIERARCHICAL PAGING algorithm, HP, is $(\gamma \cdot \text{polylog}(n) \log(\delta), 2\tau \cdot \log n \log(\delta))$-competitive for distributed paging. Since there exist $(2, 2)$-competitive uniprocessor paging algorithms (Theorem 2.1) we get the promised result.

In what follows we describe the algorithm for the read-only case. The general case follows from the discussion in the previous section. First we need to define the hierarchical graph decomposition and the distributed data structures used by algorithm HP.

4.1 Distributed Data Structures: Data Tracking

The *data tracking* mechanism of [BFR92] is a generalization of the mobile user tracking mechanism of [AP89, AP91].

In a network over a set P of n processors, the data tracking problem allows to maintain a subset Q of processors holding copies of the file with the following operations on Q:

Insert(u,v), initiated at $u \in Q$, inserts v to the set Q.

Delete(v), initiated at v, removes v from the set Q.

Find(u), initiated at u, returns the address of a processor $v \in Q$.

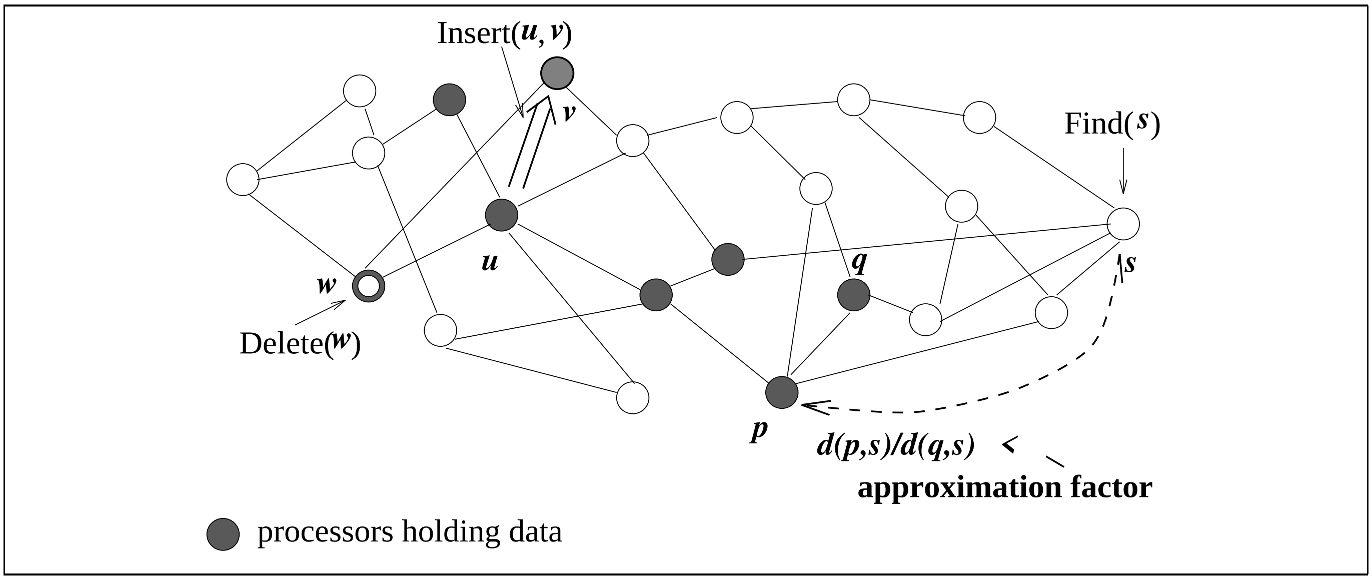

Figure 2: The operation of the data tracking mechanism.

Definition. A distributed on-line data tracking algorithm serves sequences of Insert, Delete and Find operations initiated at processors of the network. The cost of a distributed on-line data tracking algorithm for a sequence of operations is the total cost of messages it sends to conduct those operations.

The following notation is useful:

Notation. For a subset of vertices Q and a vertex p, $d(Q,p) = \min_{q \in Q}\{d(p,q)\}$.

Definition. The *approximation factor* for an on-line data tracking algorithm, α, is the maximum over all Find operations, of the ratio $d(u,v)/d(Q,u)$, where u is the node initiating the Find, and $v \in Q$ is the returned node.

Definition. The optimal cost of Insert(u,v) is the cost of transmitting the file from u to v ; i.e., $D \cdot d(u,v)$.

The optimal cost of Delete(v) is 0.

The optimal cost of Find(u) is the cost of sending a message from u to the closest processor in Q; i.e., $d(Q,u)$

[BFR92] present a distributed on-line data tracking algorithm, named TRACK, dealing with arbitrary sequences of Insert, Delete and Find operations, such that the following theorem holds.

Theorem 4.1 *For every n-processor network, for every sequence of operations σ,*

1. *TRACK's total cost for conducting Insert and Delete in* σ *is* $O(\min\{\log^2 n, \log n \log(\delta)\}/\log^2 D)$ *times the total optimal cost of those operations.*

2. *TRACK's cost on each Find in* σ *is* $O(\log^2 n/\log^2 D)$ *times the optimal cost of that Find.*

3. *TRACK's approximation factor is* $O(\log n/\log D)$.

(Where the value of D *is truncated to* $[2, n]$*).*

The data tracking competitive ratio over request sequences of of updates and searches is denoted C_{TRACK}. *We have that* $C_{\text{TRACK}} = O(\log^2 n/\log^2 D)$.

The data tracking approximation factor is denoted by α.

The memory needed for the algorithm is at most $O(\log^2(\delta))$ *per processor.*

If no memory considerations are made then the approximation factor can be reduced to $\alpha = O(1)$.

4.2 Hierarchical Graph Decompositions

The hierarchical network decomposition [AP90] defines the notion of clusters, partitions, and a hierarchy of partitions.

All these definitions are a function of some parameter k. (Usually $k = \log n$.)

A *cluster* $\mathcal{C}$ is a set of vertices.

A *partition* is a set of (possibly overlapping) clusters.

Partitions are classified into one of $\log(\delta)$ levels, denoted respectively $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{\log(\delta)}$.

The collection of these partitions is called the *hierarchical network decomposition*, denoted by $\mathcal{P}$.

A vertex may belong to several clusters and a cluster may belong to many partitions.

For every level i and every cluster $\mathcal{C} \in \mathcal{P}_i$ we arbitrarily choose a vertex $v \in \mathcal{C}$ called the i-level cluster leader and denote it by $L_i(\mathcal{C})$.

The diameter of a cluster $\mathcal{C}$, denoted by $diam(\mathcal{C})$, is defined to be the maximal distance between any two vertices in $\mathcal{C}$.

The stretch of the i'th level partition P_i is defined to be minimal upper bound S such that for every cluster $\mathcal{C} \in \mathcal{P}_i$, $diam(\mathcal{C}) \leq (2^i - 1) \cdot S$. Similarly, the stretch of a collection of partitions, $\mathcal{P}$, is defined to be the maximal stretch of any partition in $\mathcal{P}$.

The degree of an i'th level partition $\mathcal{P}_i$ is defined to be the maximal number of mutually overlapping clusters in $\mathcal{P}_i$ (i.e., clusters with nonempty intersection). Analogously, we define the degree of a collection of partitions. We use the notations $stretch(\mathcal{P}_i)$, $deg(\mathcal{P}_i)$, $stretch(\mathcal{P})$ and $deg(\mathcal{P})$ in the obvious manner.

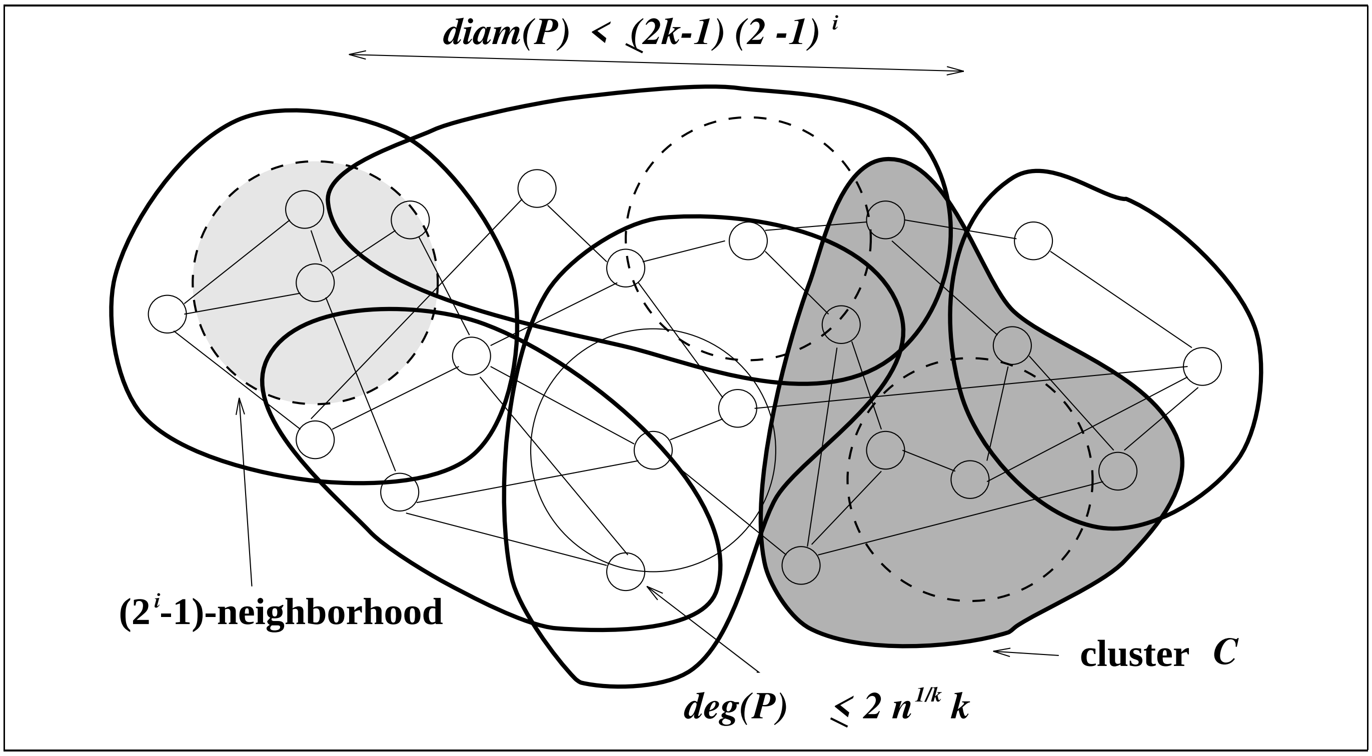

Figure 3: The i-level sparse graph partition and its properties.

Definition. The k-neighborhood of a vertex v is the set of all vertices u, s.t., $d(u, v) \leq k$. This set is denoted $N_v(k)$.

As a function of some parameter k, the level i partition, $\mathcal{P}_i$, $0 \leq i \leq \log(\delta)$, has the following properties:

- Every $2^i - 1$ neighborhood is properly contained in some cluster, $\mathcal{C} \in \mathcal{P}_i$.
- A vertex participates in at most $2 \cdot n^{1/k}k$ clusters $\mathcal{C} \in \mathcal{P}_i$, i.e., $deg(\mathcal{P}_i) \leq 2 \cdot n^{1/k}k$.
- The stretch of a cluster $\mathcal{C} \in \mathcal{P}_i$ is at most $2k - 1$, i.e., $stretch(\mathcal{P}_i) \leq 2k - 1$.

It follows from the above properties that $\mathcal{P}_0$ consists of n disjoint clusters, each of which is a single vertex.

For the case $k = \log n$ we have $deg(\mathcal{P}) \leq 2 \log n$ and $stretch(\mathcal{P}) \leq 2 \log n$.

4.3 The Hierarchical Distributed Paging Algorithm and Analysis

For a vertex v, let $\mathcal{C}_\ell(v)$ be some ℓ-level cluster such that $v \in \mathrm{Ker}_\ell(\mathcal{C}_\ell(v))$. Each cluster $\mathcal{C}$ is assigned a cluster leader $L_i(\mathcal{C})$. We let every node v know of the leader of $\mathcal{C}_\ell(v)$ for every level ℓ.

Let $\wp = deg(\mathcal{P}) \log(\delta)$. For convenience, let $\lambda = 2$.

The Hierarchical Distributed Paging Algorithm (HP).
For every level i, and every cluster $\mathcal{C} \in \mathcal{P}_i$, we define a uniprocessor paging algorithm $\mathrm{U}_i(\mathcal{C})$ for cache size equal to $\mu(\mathcal{C}) = \frac{1}{\wp} \sum_{p \in \mathcal{C}} k_p$ and the set of pages equal to the set of files $\mathcal{F}$.

HP keeps the invariant (see the following Lemma 4.3) that if F is in the cache according to $\mathrm{U}_i(\mathcal{C})$ then it is kept someplace in the cluster $\mathcal{C}$.

For every file F, the algorithm holds a read requests counter in the cluster leader $L_i(\mathcal{C})$, denoted $c_i(\mathcal{C}, F)$.

At times, the cluster leader decides to replicate a copy of a file F into the area of the cluster. For every processor $p \in \mathcal{C}$, $L_i(\mathcal{C})$ maintains the list of files assigned to it.

When a read request for file F is initiated at processor r, the request is dealt with by the leader of a cluster, containing r in its kernel, $\mathcal{C}_\ell(r)$, where ℓ is chosen such that the distance from the cluster to the nearest copy of F is proportional to the diameter of the cluster.

The reads counter maintained by that cluster leader is increased. When the counter reaches D the uniprocessor paging algorithm simulated by the cluster leader, is presented with a new request for F. Then the the cluster leader replicates and deletes files to maintain the invariant that files kept in the cache by the uniprocessor algorithm are also kept in the cluster. Appropriate data tracking operations are also performed.

Algorithm HP is given in Figure 4.

Theorem 4.2 *If* U *is (γ, τ)-competitive for uniprocessor paging then the* Hierarchical Paging *algorithm,* HP, *is $(O(\gamma \cdot \alpha \cdot \log^4 n \log(\delta)/\log^2 D), 2\tau \cdot \log n \log(\delta))$-competitive for distributed paging.*

Proof. Let $\sigma_\ell(\mathcal{C})$ be the sequence of requests assigned to $L_\ell(\mathcal{C})$. Let $\rho_\ell(C)$ be the sequence of requests generated for $\mathrm{U}_\ell(\mathcal{C})$. Along the proof, we use the set B to denote the set of processors holding a copy of file F.

The correctness of the algorithm follows from the following lemma.

Lemma 4.3 *Algorithm* HP *maintains the following invariants, for every ℓ-level cluster $\mathcal{C}$:*

1. *If file F is in the cache according to $\mathrm{U}_\ell(\mathcal{C})$ then it is kept in some processor of $\mathcal{C}$.*

2. *If a request for F is added to $\sigma_\ell(\mathcal{C})$ then prior to the request F is not kept in a processor of $\mathcal{C}$.*

3. *If no pageout occurs when $\mathrm{U}_\ell(\mathcal{C})$ deals with a request, then there exist a processor $p \in \mathcal{C}$ assigned with less than $\frac{1}{\wp}k_p$ files by $L_\ell(\mathcal{C})$.*

Proof.

1. The invariant holds since whenever $\mathrm{U}_\ell(\mathcal{C})$ deals with a request for F, F is replicated into some processor of $\mathcal{C}$.

2. Consider a read request for F initiated at processor r in $\sigma_\ell(\mathcal{C})$.

   Let s be the processor at which a copy of F has been found by HP. Let $h = d(s,r)$ be the distance of that copy to r. Let ℓ be largest such that $2^\ell - 1 \le h/(\lambda\alpha \cdot \mathit{stretch}(\mathcal{P}))$.

   Since the file copy found is at most further then the closest to r by the data tracking approximation factor we have: $h \le \alpha \cdot d(B,r)$, and therefore,

$$d(B,r) \;\; \ge \;\; \lambda \cdot \mathit{stretch}(\mathcal{P})(2^\ell - 1) \;\; > \;\; \mathit{diam}(\mathcal{C}).$$

   Since $r \in \mathcal{C}$, it follows that there is no on-line copy of F in $\mathcal{C}$.

3. If $\mathrm{U}_\ell(\mathcal{C})$ is fed with a request for a file F, then by claim 2 it is not kept in a processor of $\mathcal{C}$, and thus it follows from claim 1 that it is not in the cache according to $\mathrm{U}_\ell(\mathcal{C})$. It follows that the cache of $\mathrm{U}_\ell(\mathcal{C})$ is not filled to capacity. If for all processor $p \in \mathcal{C}$, there are at least $\frac{1}{\wp}k_p$ files assigned by $L_\ell(\mathcal{C})$ kept in p, then there are $\sum_p \frac{1}{\wp}k_p = m(\mathcal{C})$ such files overall. Every file assigned by $L_\ell(\mathcal{C})$ was paged in by $\mathrm{U}_\ell(\mathcal{C})$ and never paged out, and therefore there are $m(\mathcal{C})$ pages in its cache, a contradiction.

■

For an ℓ-level cluster $\mathcal{C}$ and a file F partition the requests in $\sigma_\ell(\mathcal{C})$ into phases of D successive requests for F. We call such a phase an $(\mathcal{C}, F, \ell)$-phase.

Consider such a phase. Let $\sigma(\mathcal{C}, F, \ell)$ be the subsequence of D requests for F issued during that phase. The last request in the $(\mathcal{C}, F, \ell)$-phase invokes a replication procedure. Let $\rho(\mathcal{C}, F, \ell)$ be the request for F issued for $\mathrm{U}_\ell(\mathcal{C})$ during this replication procedure.

We associate with every $(\mathcal{C}, F, \ell)$-phase the cost incurred by HP for serving the requests in $\sigma(\mathcal{C}, F, \ell)$ and for the replication procedure.

Lemma 4.4 *The cost associated for* HP *with every* $(\mathcal{C}, F, \ell)$*-phase obeys:*

$$\mathrm{Cost}_{\mathrm{HP}}(\sigma(\mathcal{C}, F, \ell)) \quad \leq \quad 3\lambda\alpha \cdot C_{\mathrm{TRACK}} \cdot stretch(\mathcal{P}) \cdot D \cdot (2^{\ell+1} - 1) \cdot \mathrm{Cost}_{\mathrm{U}_\ell(\mathcal{C})}(\rho(\mathcal{C}, F, \ell)).$$

Proof. Consider a read request for F initiated at processor r in $\sigma_\ell(\mathcal{C})$.

Let s be the processor at which a copy of F has been found by HP. Let $h = d(s, r)$ be the distance of that copy to r. Let ℓ be largest such that $2^\ell - 1 \leq h/(\lambda\alpha \cdot stretch(\mathcal{P}))$. Hence,

$$d(B, r) \quad \leq \quad h \quad \leq \quad \lambda\alpha \cdot stretch(\mathcal{P})(2^{\ell+1} - 1).$$

The on-line cost for serving the read is composed of two components: One is the cost of the data tracking Find operation, which is bounded by $C_{\mathrm{TRACK}} \cdot d(B, r)$. The second is the cost of communicating with the ℓ-level cluster leader of the cluster $\mathcal{C} = \mathcal{C}_\ell(r)$, containing r. The communication cost is at most $diam(\mathcal{C})$.

The $(\mathcal{C}, F, \ell)$-phase contains D such requests. For the last request, the replication procedure is invoked in some processor $p \in \mathcal{C}$, and a replication is made from s to p. Since

$$d(s, p) \quad \leq \quad d(s, r) + d(p, r) \quad \leq \quad h + diam(\mathcal{C}),$$

the cost incurred is at most $C_{\mathrm{TRACK}} \cdot D \cdot (h + diam(\mathcal{C}))$.

Thus the total on-line cost obeys,

$$\begin{aligned}\mathrm{Cost}_{\mathrm{HP}}(\sigma(\mathcal{C}, F, \ell)) \quad &\leq \quad D \cdot [C_{\mathrm{TRACK}} \cdot (d(B, r) + h + diam(\mathcal{C})) + diam(\mathcal{C})] \\ &\leq \quad 3\lambda\alpha \cdot C_{\mathrm{TRACK}} \cdot stretch(\mathcal{P}) \cdot D \cdot (2^{\ell+1} - 1).\end{aligned}$$

The lemma follows since according to Lemma 4.3, F was not in $\mathcal{C}$ prior to the replication procedure and thus not in the cache of $\mathrm{U}_\ell(\mathcal{C})$, and therefore $\mathrm{Cost}_{\mathrm{U}_\ell(\mathcal{C})}(\rho(\mathcal{C}, F, \ell)) = 1$. ∎

Let Adv be any distributed paging adversary such that for every processor p, Adv has memory capacity $h_p \leq \frac{1}{\tau\wp} k_p$ in p.

For every ℓ-level cluster $\mathcal{C}$, define a $(\mathcal{C}, \ell)$-uniprocessor paging adversary, $\mathrm{Adv}_\ell(\mathcal{C})$, as follows: at the beginning F is in the cache iff Adv keeps F in a processor of $\mathcal{C}$. Then, if F is replicated by Adv from a processor not in $\mathrm{ExKer}(\mathcal{C})$ into one in $\mathrm{ExKer}(\mathcal{C})$ then it is paged in by $\mathrm{Adv}_\ell(\mathcal{C})$.

If F in the $(\mathcal{C}, \ell)$-uniprocessor adversary cache is deleted by Adv from the last processor holding it in $\mathcal{C}$ then $\mathrm{Adv}_\ell(\mathcal{C})$ pages F out. It follows that the set of files in the cache for $\mathrm{Adv}_\ell(\mathcal{C})$, is a subset of the set of files kept in $\mathcal{C}$ by Adv. This is at most

$$\textstyle\sum_p h_p \quad \leq \quad \frac{1}{\tau\wp} \sum_p k_p \quad = \quad \frac{1}{\tau} m(\mathcal{C}),$$

files and therefore $\mathrm{U}_\ell(\mathcal{C})$ is γ-competitive against $\mathrm{Adv}_\ell(\mathcal{C})$.

We define $\mathrm{Cost}_{\mathrm{Adv}}(\sigma_\ell(\mathcal{C}))$ to be a a lower bound on the adversary cost that may be associated with a specific ℓ-level cluster $\mathcal{C}$. Every ℓ-level cluster $\mathcal{C}$ is associated with the adversary cost of serving the requests in $\sigma_\ell(\mathcal{C})$. Replications onto processors of $\mathcal{C}$ may be associated with at most $\deg(\mathcal{P})\log(\delta)$ different clusters. Every cluster can be assigned with at least a proportional fraction of the overall cost.

Let $\hat{R}(\ell) = \max\{(2^\ell - 1)/2, 1\}$.

Lemma 4.5

$$\mathrm{Cost}_{\mathrm{Adv}_\ell(\mathcal{C})}(\rho_\ell(\mathcal{C})) \cdot D \cdot \hat{R}(\ell) \quad \leq \quad 2\deg(\mathcal{P})\log(\delta) \cdot \mathrm{Cost}_{\mathrm{Adv}}(\sigma_\ell(\mathcal{C})).$$

Proof. To prove the lemma we analyze the events that cause an increase in the uniprocessor paging adversary cost:

- F is paged in/paged out by $\mathrm{Adv}_\ell(\mathcal{C})$. Between any pageout for F and the next pagein for F the adversary must have replicated F from a processor not in $\mathcal{C}$ through some sequence of processors in $\mathcal{C}$ ending at a processor in $\mathrm{ExKer}(\mathcal{C})$.

  Since the distance between a processor in $\mathrm{ExKer}(\mathcal{C})$ and a processor outside $\mathcal{C}$ is greater than $(2^\ell - 1)/2$, it follows that a cost of at least $1/(deg(\mathcal{P})\log(\delta)) \cdot D \cdot \hat{R}(\ell)$ for Adv is associated with these events.

- A page fault occurring for $\mathrm{Adv}_\ell(\mathcal{C})$ upon a request for file F, generated by $L_\ell(\mathcal{C})$ at the end of some $(\mathcal{C}, F, \ell)$-phase.

  If F was paged out by $\mathrm{Adv}_\ell(\mathcal{C})$ during the phase, then the claim follows by counting once more the cost of that pageout.

  Otherwise it follows that F was not in the cache of $\mathrm{Adv}_\ell(\mathcal{C})$ during the entire $(\mathcal{C}, F, \ell)$-phase. We therefore have that Adv did not hold a copy of F in $\mathrm{ExKer}(\mathcal{C})$ during the phase.

  Let r be the location of some read request in $\sigma(\mathcal{C}, F, \ell)$. Since $r \in \mathrm{Ker}(\mathcal{C})$, if Adv had a copy of F at a processor a at distance at most $(2^\ell - 1)/2$ from r then $N_a((2^\ell - 1)/2) \subseteq N_r((2^\ell - 1))$, and therefore $a \in \mathrm{ExKer}(\mathcal{C})$.

  It follows that the adversary cost for every request in $\sigma(\mathcal{C}, F, \ell)$ is at least $\hat{R}(\ell)$, and hence a cost of at least $D \cdot \hat{R}(\ell)$ for the entire phase.

■

Theorem 4.2 follows from Lemmas 4.3, 4.4 and 4.5, and the fact that U is γ-competitive against every $\mathrm{Adv}_\ell(\mathcal{C})$ uniprocessor paging adversary. ■

5 File Allocation on Dynamic Networks and its Applications for Distributed Paging

The distributed paging problem on arbitrary network topologies seems to be very hard. As a first step towards a solution for the problem, it would be nice to show that it is enough to deal with read requests, that is any algorithm for read-only distributed paging can be transformed into a general read-write case algorithm.

Moreover the reduction we give uses as a black box a file allocation algorithm [ABF93a].

Considerations, arising from memory constraints may conflict with file allocation strategies, this motivates the following problem:

5.1 File Allocation on Dynamic Networks

We consider the file allocation problem in a network where processors become active and inactive over time. Just before a processor goes down we assume there is enough time to replicate data stored locally elsewhere.

We give an efficient algorithm that deals with such events, and use its solution to reduce the read/write distributed paging problem to the read-only case.

Let $\mathcal{N}$ be the network of all n processors (active or inactive).

The set of active processors is a partial set of processors of $\mathcal{N}$, denoted $\mathcal{G}$.

We model the situation by the following events occurring:

- Include(p) — adds processor $p \notin \mathcal{G}$ onto $\mathcal{G}$.
- Exclude(p) — removes processor $p \in \mathcal{G}$ from $\mathcal{G}$.

Read requests occur only at the set of active processors $\mathcal{G}$, and file copies may be kept only among this set.

A processor inclusion, Include(p), is associated with an inherent cost of the distance from p to the nearest active processor in $\mathcal{G}$. A processor exclusion is associated an inherent cost of 0. Let the total inherent cost over a sequence of inclusions and exclusions be denoted INS.

We give a distributed file allocation algorithm, DynDFA, for dynamic networks that is competitive with respect to the cost of an optimal file allocation algorithm plus the inherent inclusions cost.

Theorem 5.1 *Algorithm* DynDFA *for file allocation on dynamic networks incurs a cost at most* c_{DynDFA} *times the cost of an optimal off-line file allocation algorithm for* $\mathcal{N}$ *plus* $g_{\text{DynDFA}} \cdot \text{INS}$.

Both ratios c_{DynDFA} and g_{DynDFA} are polylogarithmic in n and δ.

Before discussing the dynamic networks algorithm we first show how it can be used to obtain a competitive algorithm for the distributed paging problem.

5.2 Distributed Paging Reduces to the Read-Only Case

Let RO be an arbitrary competitive algorithm algorithm for read-only distributed paging.

Let c_{RO} be the competitive ratio of algorithm RO. We give a distributed paging algorithm, DP, for the general read-write case, by combining RO with the file allocation algorithm for dynamic networks, DynDFA.

Theorem 5.2 *Given a* c_{RO}*-competitive algorithm,* RO, *for read-only distributed paging, algorithm* DP *is* $O(c_{\text{RO}} \cdot g_{\text{DynDFA}})$*-competitive for (read-write) distributed paging.*

Distributed Paging Algorithm (DP)
Algorithm DP runs simultaneously the read-only distributed algorithm RO, and for every file, F, the file allocation algorithm for dynamic networks, DynDFA(F), for F.

The set of active processors for the dynamic network file allocation algorithm DynDFA(F) is determined by the set of processors holding copies of F according to algorithm RO.

Algorithm RO is only simulated. No actual replications or deletions for *real* files are made by RO.

That is every replication for F made by RO is translated into an Include event for the appropriate algorithm, DynDFA(F), and every deletion for F made by RO is translated into an Exclude event for DynDFA(F).

DP invokes, upon every request, both algorithms RO and DynDFA in the following manner:

Upon receiving a *read*(F) request at processor p: let q be the processor such that RO will read from given a read request for F at p. Note that q is currently an active processor for DynDFA. Initiate a *read*(F) request at q for algorithm DynDFA(F). The data read by q is transmitted to p. Next, initiate a *read*(F) at p for algorithm RO. Initiate appropriate Include/Exclude operations for DynDFA for every replication/deletion made by RO.

Upon receiving a *write*(F) request at processor p, let q be the processor such that RO will read from given a read request for F at p. Note that q is currently an active processor for DynDFA. Initiate a *write*(F) request at q for algorithm DynDFA(F). Next, initiate a *read*(F) at p for algorithm RO. Initiate appropriate Include/Exclude operations for DynDFA for every replication/deletion made by RO.

The following 2 lemmas and Theorem 5.1 give the proof of Theorem 5.2.

Let σ be an arbitrary sequence of read and write requests.

For a file F, let $\sigma(F)$ be the subsequence of read and write requests issued for F in σ. Let $\sigma'(F)$ be the sequence of read and write requests for F initiated for algorithm DynDFA(F) by DP, and all Include/Exclude commands initiated by DP.

Let σ_{RO} be the sequence obtained from σ by replacing every write request with a read request for the same file at the same location. Also let $\sigma_{\mathrm{RO}}(F)$ be the subsequence of read requests issued for F in σ_{RO}.

The cost paid by DP is the cost of communicating between p and the nearest processor to p such that RO holds a copy of F, q, plus the work performed by DynDFA. If a read request is handled then another communication between q and p is performed.

Lemma 5.3

$$\mathrm{Cost}_{\mathrm{DP}}(\sigma) \quad \leq \quad \mathrm{Cost}_{\mathrm{RO}}(\sigma_{\mathrm{RO}}) + \sum_{F\in\mathcal{F}} \mathrm{Cost}_{\mathrm{DynDFA}(F)}(\sigma'(F)).$$

Given any distributed paging adversary Adv, define an adversary for read-only paging $\mathrm{Adv}_{\mathrm{RO}}$ that maintains the same configuration as Adv does. Similarly, for every file F, define a file allocation adversary Adv(F) that keeps the file F at the same processors as Adv does.

Lemma 5.4

$$\mathrm{Cost}_{\mathrm{Adv}_{\mathrm{RO}}}(\sigma_{\mathrm{RO}}) \quad \leq \quad \mathrm{Cost}_{\mathrm{Adv}}(\sigma), \tag{1}$$

$$\mathrm{Cost}_{\mathrm{Adv}(F)}(\sigma'(F)) \quad \leq \quad \mathrm{Cost}_{\mathrm{Adv}}(\sigma(F)) + \mathrm{Cost}_{\mathrm{RO}}(\sigma_{\mathrm{RO}}(F)). \tag{2}$$

Proof. Let the set of processors holding a file F, be $A(F)$.

The cost for a *read*(F) issued at p is the distance from p to the nearest copy of F: $d(A(F),p)$.

The cost for a *write*(F) issued at the same position, equals the weight of the minimum Steiner tree spanning p and all copies of F: $T(A(F)\cup\{p\}) \geq d(A(F),p)$.

It follows that $\mathrm{Cost}_{\mathrm{Adv}_{\mathrm{RO}}}(\sigma_{\mathrm{RO}}) \leq \mathrm{Cost}_{\mathrm{Adv}}(\sigma)$.

For a request for F issued at p let q be the processor from which algorithm RO reads in response to the *read*(F) request issued by DP for RO. Thus the cost for RO for this request is at least $d(p,q)$.

For a *read*(F) request, a read request is issued for algorithm DynDFA(F) at q. The cost for the adversary for this request is

$$d(A(F),q) \ \leq \ d(A(F),p) + d(p,q).$$

For a *write*(F) request, a write request is issued for algorithm DynDFA(F) at q. The cost for the adversary for this request is

$$T(A(F)\cup\{q\}) \ \leq \ T(A(F)\cup\{p\}) + d(p,q).$$

In both cases the cost for Adv(F) is at most the adversary cost for the request at p plus the cost for RO for the read request at p. ■

Proof of Theorem 5.2. Combining Lemmas 5.3 and 5.4 with Theorem 5.1 we obtain:

$$\begin{aligned}
\mathrm{Cost}_{\mathrm{DP}}(\sigma) &\leq \mathrm{Cost}_{\mathrm{RO}}(\sigma_{\mathrm{RO}}) + \sum_{F\in\mathcal{F}} \mathrm{Cost}_{\mathrm{DynDFA}(F)}(\sigma'(F)) \\
&\leq \mathrm{Cost}_{\mathrm{RO}}(\sigma_{\mathrm{RO}}) + \sum_{F\in\mathcal{F}} (g_{\mathrm{DynDFA}} \cdot \mathrm{Cost}_{\mathrm{Adv}(F)}(\sigma'(F)) + a_0) \\
&\leq \mathrm{Cost}_{\mathrm{RO}}(\sigma_{\mathrm{RO}}) + g_{\mathrm{DynDFA}} \cdot (\mathrm{Cost}_{\mathrm{Adv}}(\sigma) + \mathrm{Cost}_{\mathrm{RO}}(\sigma_{\mathrm{RO}})) + |\mathcal{F}|\cdot a_0 \\
&= g_{\mathrm{DynDFA}} \cdot \mathrm{Cost}_{\mathrm{Adv}}(\sigma) + (g_{\mathrm{DynDFA}}+1)\cdot \mathrm{Cost}_{\mathrm{RO}}(\sigma_{\mathrm{RO}}) + |\mathcal{F}|\cdot a_0,
\end{aligned}$$

where a_0 is some additive term.

Using the assumption on the competitiveness of algorithm RO, we get

$$\begin{aligned}
\mathrm{Cost}_{\mathrm{DP}}(\sigma) &\leq g_{\mathrm{DynDFA}} \cdot \mathrm{Cost}_{\mathrm{Adv}}(\sigma) + (g_{\mathrm{DynDFA}}+1)\cdot (c_{\mathrm{RO}} \cdot \mathrm{Cost}_{\mathrm{Adv_{RO}}}(\sigma_{\mathrm{RO}}) + a_1) + |\mathcal{F}|\cdot a_0 \\
&\leq (g_{\mathrm{DynDFA}} \cdot c_{\mathrm{RO}} + g_{\mathrm{DynDFA}} + c_{\mathrm{RO}}) \cdot \mathrm{Cost}_{\mathrm{Adv}}(\sigma) + |\mathcal{F}|\cdot a_0 + (g_{\mathrm{DynDFA}}+1)c_{\mathrm{RO}} \cdot a_1,
\end{aligned}$$

for some additive terms a_0 and a_1.

This concludes the theorem. ■

We next turn to describe the solution for the file allocation problem on dynamic networks. The basic ideas in the way the algorithm deals with inclusions and exclusions is motivated by the vertex inclusion/exclusion problem.

5.3 The Vertex Inclusion-Exclusion Problem

In this section we define a new natural combinatorial problem we call the *vertex inclusion-exclusion* problem.

The theorem presented here is useful in analyzing problems in a dynamic network as demonstrated in Section 5.

Let G be an underlying weighted graph. Let Q be a subset of nodes of G.

Given a sequence of insertions for vertices of G into Q, and deletions for nodes in Q, we associate with each operations an inherent cost.

The cost of an insert operation is equal to the distance from the inserted node to the nearest node in Q.

The cost of a delete operation is equal to the distance from the deleted node to the nearest remaining node in Q.

Note that this last definition is different from the usual "inherent" cost of zero of a delete, which we usually use.

The reason is that our goal is in fact to compare the total deletions cost to the total insertions cost, and thus in a way, justifying the deletion cost defined above.

The following theorem is a consequence of the hierarchical cover problem solution properties (Section 5.3).

Theorem 5.5 *Consider a vertex inclusion-exclusion problem. For any sequence of insertions and deletions, let* INS *denote the total insertions cost and let* DEL *denote the total deletions cost. Then*

$$\mathrm{DEL} \quad \leq \quad O(\min\{\log n, \log(\delta)\}) \cdot \mathrm{INS}.$$

To prove theorem 5.5 we use make use of another basic combinatorial problem defined in [BFR92] called the *cover problem*.
The Cover Problem. The k-cover problem is that of maintaining a partition of a set of vertices Q of a graph G, into s disjoint non empty covering subsets of Q such that the diameter of each subset at most k. The set Q is allowed to change dynamically under a sequence of vertex insertions and deletions.

The optimal cost of an insertion, is defined in the natural way, to be the distance between the origin node p and the inserted node q, and the optimal cost for a deletion to be 0.

The *hierarchical cover problem* is that of maintaining simultaneously $(2^{i+1}-1)$-cover problems for all levels $0 \leq i \leq \log(\delta)$.

Let the number of of covering sets maintained by the i-level cover algorithm be denoted $s(i)$.

Let INS denote the total optimal cost over a sequence of insertions and deletions. The following theorem is due to [BFR92]:

Theorem 5.6 *The following holds for the hierarchical cover algorithm*

$$\sum_{i=0}^{\log(\delta)} 2^i \cdot (s(i)-1) \quad \leq \quad \sum_{i=0}^{\log(\delta)} 2^i \cdot (c(i)-1) \quad \leq \quad O(\min\{\log n, \log(\delta)\}) \cdot \mathrm{INS}.$$

Proof of Theorem 5.5. Define a hierarchical cover problem for the dynamically changing set Q in the natural way; I.e., each insertion is done from a the nearest node in Q to the inserted node, and each deletion is initiated at the deleted node. The inherent cost for the cover problem defined is just the insertions cost INS.

Let the number of of covering sets maintained by the i-level cover algorithm be denoted $s(i)$. Let the total number of covering sets creations made by the i-level cover algorithm be denoted $c(i)$. Then $s(i) \leq c(i)$.

Consider the potential function

$$\Phi = \sum_{i=0}^{\log(\delta)} 2^{i+1} \cdot (c(i) - s(i)).$$

It follows from Theorem 5.6 that $\Phi \leq O(\min\{\log n, \log(\delta)\}) \cdot \mathrm{INS}$.

Now, consider a deletion of node $v \in Q$ and let u be the nearest node to v in $Q \setminus \{v\}$. The deletions cost, DEL, increases by $d(u, v)$.

Let i be the maximal such that $2^i < d(u, v)$. Consider the $(i-1)$-level covering set C containing v just before the deletion of v. Since the diameter of this subgraph is at most 2^i, v is the only vertex in Q contained in the covering set C, and therefore $s(i-1)$ will decrease by 1 when v is deleted, increasing the potential Φ by at least $d(u, v)/2$.

Since $s(i) \leq c(i)$ then $\Phi \geq 0$. We conclude that

$$\mathrm{DEL} \quad \leq \quad 2\Phi \leq O(\min\{\log n, \log(\delta)\}) \cdot \mathrm{INS}.$$

■

5.4 The Solution for File Allocation on Dynamic Networks

The file allocation algorithm on *dynamic networks*, DynDFA, is a modification of the distributed file allocation algorithm, DFA, of [ABF93a], designed to deal with inclusions and exclusions of active processors.

The basic file allocation algorithm, DFA, has the property that when file copies are replicated, the replication is always made to the location of the just arrived read request and therefore to a location of an active processor (in $\mathcal{G}$).

The DynDFA algorithm maintains a data tracking mechanism for the set $\mathcal{G}$ of active processors. When a processor, p, holding a copy of the file receives an Exclude(p) event, algorithm DynDFA finds a processor, q, close to p in $\mathcal{G}$ using the data tracking mechanism and replicates the file copy to that processor.

It follows from Theorem 5.5 for the vertex inclusion/exclusion problem that the cost incurred by the algorithm for the communication from p to q (the exclusions cost) can be amortized against the cost of processor inclusions up to a $\log n$ times the data tracking approximation factor. We use a potential function proof to show that a similar argument also bounds the damage that may be caused by the exclusion of p to the file allocation strategy.

We now describe the dynamic network file allocation algorithm, DynDFA, in details.

The algorithm uses some distributed data structures additional to the graph decompositions and data tracking mechanism that are described below.

5.4.1 Additional Distributed Data Structures.

Scanning Mechanism.

Another simple tool is a distributed data structure that enables scanning through a subset of processors.

The scanning mechanism is usually used together with the data tracking mechanism, as to enable performing tracking operations over all processors, such as a Delete-All operation.

In a network over a set P of n processors, let Q be a subset of processors in P. Initially Q includes a single processor v_0.

Given a sequence of insertions for processors $v_1, v_2, \ldots, v_t$, we maintain a distributed data structure.

The processors $v_1, v_2, \ldots, v_t$ are connected in a tree structure $\mathcal{T}$, by using an adjacency list at each processor in $\mathcal{T}$.

When an insertion for v_i arrives: i.e., Insert(v_j,v_i) , $j < i$ is made, we add v_i to the tree $\mathcal{T}$, by leaving a pointer in v_j to v_i, we also leave a back-tracking pointer in v_i to v_j.

This procedure enables scanning through all processors in the tree structure starting at every one of the processors in $\mathcal{T}$, at a cost equal to the weight of $\mathcal{T}$.

The above description requires a considerable amount ($\Theta(n)$) of memory per processor, for keeping the list of its children. In the next section we describe how this can be reduced to $O(\log(\delta))$. **Lists Manipulation.**

We now turn to the question of memory needs of our distributed algorithms. In our algorithms processors, p, maintain lists of pointers $L(p)$ to other processors.

Assume that the address of a processor appears in the lists of at most x processors.

Keeping every list at the processor maintaining it requires $\Theta(n)$ pointers per processor.

To overcome this difficulty, for every processor we translate the list $L(p)$, into $\log(\delta)$ lists. List $L_i(p)$, $0 \leq i \leq \log(\delta)$, is the list of processors at distance between 2^i and $2^{i+1} - 1$ from p.

Now, instead of keeping the entire list in p, pointers are kept to only one of the processors in each of the $\log(\delta)$ lists, and within a list $L_i(p)$, each processor in the list keeps a pointer to the next. The distance between any two processors u and v in $L_i(p)$ is at most $d(u,p) + d(p,v) \leq 2(2^{i+1} - 1)$, and therefore every operation on the list $L_i(p)$, is proportional to 2^i.

It follows that every processor needs only $\log(\delta)$ pointers for maintaining the information for its own list, and since it may appear in the lists of at most x processors it may need x more pointers. Therefore the total memory requirements is reduced to $O(x + \log(\delta))$ pointers per processors.

The operations provided by this data structures, for the list $L(p)$, are:

- Insert(v) — insert the address of processor v into $L(p)$, this is done by updating the pointers in the required level. The cost associated is $O(d(p,v))$.
- Delete(v) — delete the address of processor v into $L(p)$, this is done by updating the pointers in the required level. The cost associated is $O(d(p,v))$.
- Find — Find a processor in $L(p)$, by using the pointer to a processor in the nonempty list $L_i(p)$ with minimal i. The cost associated is $O(d(p, L(p)))$.
- Scan — Scan the entire list, at a cost proportional to the sum of communication costs between p and every processor in $L(p)$.

The distributed data structures used by algorithm DynDFA are:

- The hierarchical graph decomposition (Section 4.2). The network is decomposed in $\log(\delta)$ levels partitions $\mathcal{P} = \{\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{\log(\delta)}\}$, each of which partitions the graph into clusters, such that every $(2^i - 1)$-neighborhood is contained in an i-level cluster, and $stretch(\mathcal{P}) \leq 2\log n$ and $deg(\mathcal{P}) \leq 2\log n$.

Definition. The *kernel* of an i-level cluster $\mathcal{C} \in \mathcal{P}_i$ is the set of vertices such that their $(2^i - 1)$-neighborhood is contained in $\mathcal{C}$. The kernel of an i-level cluster $\mathcal{C}$ is denoted $\mathrm{Ker}_i(\mathcal{C})$.

Similarly define the *extended-kernel* of an i-level cluster $\mathcal{C} \in \mathcal{P}_i$ to be the set of vertices such that their $(2^i - 1)/2$-neighborhood is contained in $\mathcal{C}$. The extended-kernel of an i-level cluster $\mathcal{C}$ is denoted $\mathrm{ExKer}_i(\mathcal{C})$.

It follows that for every vertex v, there exists at least one cluster $\mathcal{C} \in \mathcal{P}_i$, such that $v \in \mathrm{Ker}_i(\mathcal{C}) \subseteq \mathrm{ExKer}_i(\mathcal{C})$. For a vertex v, let $\mathcal{C}_\ell(v)$ be some ℓ-level cluster such that $v \in \mathrm{Ker}_\ell(\mathcal{C}_\ell(v))$.

Each cluster $\mathcal{C}$ is assigned a cluster leader $L_i(\mathcal{C})$. We let every node v know of the leader of $\mathcal{C}_\ell(v)$ for every level ℓ.

- Data tracking (Section 4.1) and scanning mechanism (Section 5.4.1), for the set Q of processors holding copies of the file.

- Data tracking mechanism for the set $\mathcal{G}$ of active processors. To distinguish between the operations of this data tracking mechanism and the former, we add the set $\mathcal{G}$ before the operation. (I.e., $\mathcal{G}$-Insert, $\mathcal{G}$-Delete, $\mathcal{G}$-Find.)

  Let $C_{\text{TRACK}} = O(\log^2 n / \log^2 D)$ be the data tracking competitive ratio, and let α be the approximation factor of the data tracking mechanism.

- To save memory requirements, lists of processor addresses are divided into $\log(\delta)$ separate lists for processors of about the same distance from the node maintaining the list.

We assume that initially the algorithm holds a single copy of the file in the network. Otherwise we can *delete* all copies but one, incurring no additional cost. Let $\lambda = 2$.

Distributed File Allocation Algorithm for Dynamic Networks (DynDFA)
We first describe how the algorithm deals with processors inclusions and exclusions.

Dealing with Inclusions. Upon an Include(p) event we initiate a data tracking $\mathcal{G}$-Insert(p) operation.

Dealing with Exclusions. Upon receiving an Exclude(p) event, if p holds a file copy then we initiate a data tracking $\mathcal{G}$-Delete(p) operation. We then initiate a $\mathcal{G}$-Find(q) operation to find a processor q, in $\mathcal{G}$, and replicate the file from p onto q, and initiate an Insert(p,q) followed by a Delete(p).

The algorithm partitions the request sequence into phases. Each phase, except perhaps for the last one, contains exactly D consecutive *write* requests.

Dealing with read requests. For every level, i, and every cluster $\mathcal{C} \in \mathcal{P}_i$, the algorithm holds a read requests counter in the cluster leader $L_i(\mathcal{C})$, denoted $c_i(\mathcal{C})$.

Every processor keeps a list of processors reading the file from it during the phase.

Initially all counters, $c_i(\mathcal{C})$, are set to zero.

Upon receiving a new *read* request initiated by processor r, we search for a copy of the file using the distributed data tracking mechanism's Find(r) operation.

Let s be the processor at which a copy of the file has been found. s updates the list of processors reading it to include r. Let $h = d(s, r)$ be the distance of that copy to r.

Let ℓ be largest such that for $k = 2^\ell - 1$, $k \leq h/(\lambda\alpha \cdot stretch(\mathcal{P}))$.

Let $\mathcal{C} = \mathcal{C}_\ell(r)$ be the predetermined ℓ-level cluster such that $r \in \mathrm{Ker}(\mathcal{C})$. Inform cluster leader $L_\ell(\mathcal{C})$ to increase the read counter $c_\ell(\mathcal{C})$ by 1.

If the counter, $c_\ell(\mathcal{C})$, reached D, then initialize the counter to 0. Then, *replicate* a copy of the file to r from processor s, holding a copy of the file.

Update the distributed data structures, by initiating an Insert(s,r) data tracking operation, and performing the same insertion to the tree scanning mechanism, $\mathcal{T}$.

Dealing with write requests. Algorithm DynDFA deals with writes exactly the same as algorithm DFA.

A list of the locations of the write requests issued during the phase (with repetitions), is kept at the processor initially holding a copy of the file, denoted z.

The algorithm maintains the tree scanning mechanism, $\mathcal{T}$, spanning the set of copies it holds by adding to it the edges along which it replicates.

When a *write* request arrives at w, DynDFA initiates a Find(w) operation to find a processor holding a copy of the file, then it traverses the tree $\mathcal{T}$, updating the data in all processors holding a copy of the file, adds the location w of the writing processor to the list at the processor, z, initially holding a copy of the file.

Once the write counter at z reaches the file size D, the phase ends, and we make a "*reorganization step*", as follows:

Let the D locations where the file has been written be $w_1, w_2, \ldots, w_D$. Find the "central" location, $m = w_i$, with the property that its distance from all other $D-1$ locations is minimized.

The central location, m, is computed by processor z, that knows about all the D write locations.

Then a replication is made onto m, by performing a Find(m) operation and then replicating a copy from the found location, s. Update the data tracking structure by initiating Insert(s,m).

Now, DynDFA traverses the tree $\mathcal{T}$ again, for every processor in $\mathcal{T}$, communicates with every processor in the list of processors reading from it to inform all cluster leaders defined in the read procedure above to initialize their read counters. Then DynDFA deletes all the replicas , except for the last one at m.

For each deleted replica kept at processor v, update the distributed data structure by initiating a Delete(v) data tracking operation, and eventually initialize the tree scanning mechanism $\mathcal{T}$ to the single node m.

Let $c_{\mathrm{DynDFA}} = O(\alpha \cdot \log^5 n / \log^2 D)$. and let $g_{\mathrm{DynDFA}} = c_{\mathrm{DynDFA}} \alpha \log(\delta)$.

Theorem 5.1 Algorithm DynDFA for file allocation on dynamic networks incurs a cost at most c_{DynDFA} times the cost of an optimal off-line file allocation algorithm for $\mathcal{N}$ plus $g_{\mathrm{DynDFA}} \cdot \mathrm{INS}$.

We use the *vertex inclusion-exclusion* theorem from Section 5.3 to account for the on-line cost incurred for the replications made upon processor exclusions.

Proof of Theorem 5.1:

We prove the competitiveness of algorithm DynDFA using a potential function argument. We analyze the change in the potential function in response to requests issued and the corresponding operations taken by the algorithm and adversary.

In addition we prove that the increase in the potential function upon the actions made in response to exclusions is bounded by $O(\alpha \log(\delta) \cdot \log^3 n / \log D)$ times DEL, where DEL is the total inherent cost of exclusions according to the vertex inclusion-exclusion problem; i.e., the sum over all exclusions, of the distance to the processor left in $\mathcal{G}$, nearest to the removed processor.

We then use Theorem 5.5 for the vertex inclusion-exclusion problem, which implies that DEL $\leq$

$\min\{\log n, \log(\delta)\} \cdot \mathrm{INS}$, to obtain the theorem.

Let B denote the online configuration, and $\overline{B}$ the subtree implied by the on-line replications. Let $\overline{A}$ denote the subtree implied by the intermediate adversary replications.

We define a *hierarchical tree cover* (Section 5.3) of the intermediate adversary subtree $\overline{A}$.

For every level $0 \leq i \leq \log(\delta)$ we have a tree cover of $s(i)$ covering subtrees of diameter at most $2(2^i - 1)$.

Let $p^i_1, p^i_2, \ldots, p^i_{s(i)}$ be the covering nodes in the i-level tree cover.

Let $\beta = \lambda \cdot stretch(\mathcal{P}) \cdot (\alpha \cdot C_{\mathrm{TRACK}} + 1)$.

Let $0 \leq \ell \leq \log(\delta)$. Let $\lambda^*_1 = 8\beta$, and $\lambda^* = 2\lambda^*_1$.

Let $\tilde{R}(\ell) = \max\{(2^\ell - 1), 1\}$ and $R(\ell) = \lambda \cdot stretch(\mathcal{P}) \cdot (2^\ell - 1)$. Let $i(\ell) = \max\{\ell - 3, 0\}$. For every ℓ-level cluster, $\mathcal{C}$, then if there exist $i(\ell)$-level covering processors, $p_j^{i(\ell)}$, such that $p_j^{i(\ell)}$ is in the extended-kernel of $\mathcal{C}$, then a credit proportional to the distance between the kernel of $\mathcal{C}$ and the set of on-line file copies B is given, as long as it is at most $\tilde{R}(\ell)$.

$$\Psi = \lambda^* \cdot D \sum_{\ell=0}^{\log(\delta)} \sum_{\mathcal{C} \in \mathcal{P}_\ell} \begin{cases} 0 & \text{if } \forall_{j=1}^{s(i)} p_j^{i(\ell)} \notin \mathrm{ExKer}(\mathcal{C}) \\ d_{\tilde{R}(\ell)}(B, \mathrm{Ker}(\mathcal{C})) & \text{otherwise} \end{cases}$$

Similarly, we define Λ. For every level ℓ and cluster $\mathcal{C} \in \mathcal{P}_\ell$, let $e_\ell(\mathcal{C})$ be the number of read requests, issued during the current phase, that have been dealt with in $\mathcal{C}$; i.e., requests that have increased the counter $c_\ell(\mathcal{C})$ during the execution of the algorithm in the current phase.

$$\Lambda = \lambda^*_1 \cdot e_\ell(\mathcal{C}_\ell(p_j^{i(\ell)})) \sum_{\ell=0}^{\log(\delta)} \sum_{\mathcal{C} \in \mathcal{P}_\ell} \begin{cases} 0 & \text{if } \forall_{j=1}^{s(i)} \mathcal{C}_\ell(p_j^{i(\ell)}) \neq \mathcal{C} \\ d_{\tilde{R}(\ell)}(B, \mathrm{Ker}(\mathcal{C})) & \text{otherwise} \end{cases}$$

Finally, let $\Theta = C_{\mathrm{TRACK}} \cdot D \cdot T(\overline{B})$, the potential function is:

$$\Phi = \Psi - \Lambda + \Theta.$$

5.1.2 The Potential Function Analysis.

Lemma 5.7 *The cost expended for data tracking update operations over the set $\mathcal{G}$ of active processors is at most $C_{\mathrm{TRACK}} \cdot \mathrm{INS}$.*

Proof. The lemma follows directly from the data tracking mechanism properties. ■

Lemma 5.8 *The total change in Φ over the sequence of processor exclusion events as well as the total cost incurred for dealing with the exclusions are at most $O(\alpha\beta \cdot \log(\delta) \cdot deg(\mathcal{P})) \cdot \mathrm{DEL}$.*

Using Theorem 5.5 (vertex inclusion-exclusion theorem) we have that this is at most $O(\alpha\beta \cdot \log(\delta) \cdot deg(\mathcal{P}) \cdot \min\{\log n, \log(\delta)\}) \cdot \mathrm{INS}$.

Proof. Consider an Exclude(p) event, for some $p \in \mathcal{G}$. If p does not hold a copy of the file then no operations are taken, and no cost incurred.

If p holds a file copy then it is deleted from the set of active processors, and a $\mathcal{G}$-Find(q) operation is generated to find a processor q close to p, and a file copy is replicated from p onto q.

The inherent cost for the exclusion is defined to be $d(\mathcal{G}, p)$.

Let $h = d(p, q)$. Then by the approximation factor property of the data tracking mechanism we have that $h \leq \alpha \cdot d(\mathcal{G}, p)$.

The cost incurred for the $\mathcal{G}$-Find operation is at most $C_{\mathrm{TRACK}} \cdot d(\mathcal{G}, p)$, and the cost for the Insert(p,q) is at most $C_{\mathrm{TRACK}} \cdot h$.

It follows that the total on-line cost associated with the management of the Exclude(p) operation obeys

$$\mathrm{Cost}_{\mathrm{DynDFA}} \;\leq\; \alpha \cdot C_{\mathrm{TRACK}} \cdot d(\mathcal{G}, p).$$

We now turn to analyze the change that may be caused to Φ as the result of the removal of p from B and the insertion of q onto B.

The increase in Θ for the replication onto q is at most $d(p, q)$. The deletion at p does not affect Θ, and therefore,

$$\Delta\Theta \;\leq\; \alpha \cdot d(\mathcal{G}, p).$$

The replication onto q may not increase $\Psi - \Lambda$, so that we need consider only the deletion at p.

For an ℓ-level cluster $\mathcal{C}$, the value of $d_{\tilde{R}(\ell)}(B, \mathrm{Ker}(\mathcal{C}))$ may increase only if $d(\mathrm{Ker}(\mathcal{C}), p) < \tilde{R}(\ell)$.

For $\ell = 0$, $\mathcal{C}$ is a single node, and this means that $\mathcal{C} = \{p\}$.

Otherwise, $d(\mathrm{Ker}(\mathcal{C}), p) < 2^{\ell} - 1$ and from the definition of the cluster kernel we get that $p \in \mathcal{C}$. Therefore there are at most $deg(\mathcal{C})$ such clusters for a fixed level ℓ.

The increase in $d_{\tilde{R}(\ell)}(B, \mathrm{Ker}(\mathcal{C}))$ is bounded by $d(p, q) \leq \alpha \cdot d(\mathcal{G}, p)$.

Hence the total change in $\Psi - \Lambda$ obeys

$$\Delta\Psi - \Delta\Lambda \quad \leq \quad \lambda^* \cdot \log(\delta) \cdot \deg(\mathcal{P}) \cdot \alpha \cdot d(\mathcal{G}, p). \blacksquare$$

Lemma 5.9 *The potential function, Φ, is nonnegative.*

Proof. We prove that $\Lambda \leq \Psi$.

Consider some level ℓ and cluster $\mathcal{C} \in \mathcal{P}_\ell$.

It follows from the definition of the algorithm that whenever a read request, initiated at some processor, r, increases the counter $c_\ell(\mathcal{C})$ for the cluster, then $r \in \mathcal{C}$, and if $c_\ell(\mathcal{C})$ reaches a count of D then a file is replicated onto r, and hence onto $\mathcal{C}$. The counter is set to 0 and will not be increased again till the next phase arrives.

Therefore, we have

$$e_\ell(\mathcal{C}) \quad \leq \quad c_\ell(\mathcal{C}) \quad \leq \quad D.$$

Since $\lambda^* \geq \lambda_1^*$ we infer that $\Lambda \leq \Psi$. $\blacksquare$

Lemma 5.10 *The total change in Φ due to adversary replications and deletions during a phase is at most $O(\beta \cdot deg(\mathcal{P}) \cdot \min\{\log n, \log(\delta)\})$ times the cost of these operations.*

Proof. Since for every covering processor there are at most one $deg(\mathcal{P})$ clusters that contain it in their extended-kernel, it follows from the definition of the potential function that

$$\Psi \quad \leq \quad \lambda^* \cdot deg(\mathcal{P}) \cdot D \sum_{\ell=0}^{\log(\delta)} \sum_{j=1}^{s(i(\ell))} \tilde{R}(\ell)$$

The proof follows from theorem 5.6. $\blacksquare$

Lemma 5.11 *For each read request, Φ increases by at most $O(\beta)$ times the adversary cost for that request minus the cost* DynDFA *incurs for the same request.*

Proof. Consider a read request initiated at processor r.

Let s be the processor at which a copy of the file has been found by DynDFA. Let $h = d(s, r)$ be the distance of that copy to r.

Let ℓ be largest such that $2^\ell - 1 \leq h/(\lambda\alpha \cdot stretch(\mathcal{P}))$.

Since the file copy found is at most further then the closest to r by the data tracking approximation factor we have: $h \leq \alpha \cdot d(B, r)$, and hence,

$$\lambda \cdot stretch(\mathcal{P})(2^\ell - 1) \quad \leq \quad d(B, r) \quad \leq \quad \lambda\alpha \cdot stretch(\mathcal{P})(2^{\ell+1} - 1),$$

that is $R(\ell) \leq d(B,r) \leq \alpha \cdot R(\ell+1)$.

The on-line cost for serving the read is composed of two components: One is the cost of the data tracking Find operation, which is bounded by $C_{\text{TRACK}} \cdot d(B,r)$.

The second is the cost of communicating with the ℓ-level cluster leader of the cluster $\mathcal{C} = \mathcal{C}_\ell(r)$, containing r. The communication cost is at most $diam(\mathcal{C}) \leq (2^\ell - 1) \cdot stretch(\mathcal{P})$.

Thus the total on-line cost obeys,

$$\begin{aligned} \text{Cost}_{\text{DynDFA}} &\leq C_{\text{TRACK}} \cdot d(B,r) + R(\ell) \\ &\leq (\alpha \cdot C_{\text{TRACK}} + 1) \cdot R(\ell+1) \\ &\leq \beta \cdot (2^{\ell+1} - 1), \end{aligned} \tag{3}$$

using $R(\ell) \leq d(B,r) \leq \alpha \cdot R(\ell+1)$.

If the intermediate adversary does not hold a copy of the file at distance smaller than $(2^\ell - 1)/4$ from r then the real adversary does no t hold such a copy either and therefore must expend a cost of $\text{Cost}_{\text{Adv}} \geq \max\{(2^\ell - 1)/4, 1\}$ upon the request.

A read request cannot cause an increase in the potential function, and thus

$$\Delta\Phi \;\leq\; 0 \;=\; 2\beta \cdot (4 \cdot \tfrac{2^\ell - 1}{4} + 1) - \beta \cdot 2^{\ell+1} \;\leq\; 10\beta \cdot \text{Cost}_{\text{Adv}} - \text{Cost}_{\text{DynDFA}}.$$

Otherwise, the intermediate adversary holds a file copy at a processor a at distance less than $(2^\ell - 1)/4$ from r. Let $i(\ell) = \max\{\ell - 3, 0\}$, then there exists a covering processor p at the $i(\ell)$-level tree cover at distance of at most $\max\{2(2^{\ell-3} - 1), 0\} \leq (2^\ell - 1)/4$ from a. Therefore,

$$d(p,r) \;\leq\; d(p,a) + d(a,r) \;\leq\; (2^\ell - 1)/4 + (2^\ell - 1)/4 \;\leq\; (2^\ell - 1)/2.$$

Let $\mathcal{C}$ be a cluster such that the read request increases its read counter. By the algorithm definition $r \in \text{Ker}(\mathcal{C})$.

It follows that the $(2^\ell - 1)/2$-neighborhood of p is contained in the $(2^\ell - 1)$-neighborhood of r and therefore $p \in \text{ExKer}(\mathcal{C})$.

$$\begin{aligned} d(B, \text{Ker}(\mathcal{C})) &\geq d(B,r) - \max_{q \in \text{Ker}(\mathcal{C})} d(q,r) \\ &> R(\ell) - (2^\ell - 1) stretch(\mathcal{P}) \\ &= (\lambda - 1) stretch(\mathcal{P}) \cdot (2^\ell - 1). \end{aligned}$$

Therefore,

$$d(B, \text{Ker}(\mathcal{C})) \;\geq\; \max\{(2^\ell - 1), 1\} \;=\; \tilde{R}(\ell).$$

We now turn to analyze the change in Λ, in response to the request at r. It follows that,

$$\Delta\Lambda \;\geq\; \lambda_1^* \cdot d_{\tilde{R}(\ell)}(B, \mathrm{Ker}(\mathcal{C})) \;\geq\; \lambda_1^* \cdot \tilde{R}(\ell) \;\geq\; \beta \cdot 2^{\ell+1}$$

Therefore, using inequality 3 we conclude that

$$\Delta\Phi \;\leq\; -\Delta\Lambda \;\leq\; -\mathrm{Cost}_{\mathrm{DynDFA}}.$$

■

Lemma 5.12 *For each replication* DynDFA *performs,* Φ *increases by at most* $O(\beta)$ *times the adversary cost for the* D *read requests associated with the replication minus the cost incurred by* DynDFA *for the replication.*

Proof. Let r be the processor DynDFA replicated to. Let B be the on-line configuration prior to the replication.

Let s be the processor at which a copy of the file has been found by DynDFA. Let $h = d(s, r)$ be the distance of that copy to r.

Let ℓ be largest such that $2^\ell - 1 \leq h/(\lambda\alpha \cdot stretch(\mathcal{P}))$.

Since the file copy found is at most further then the closest to r by the data tracking approximation factor we have: $h \leq \alpha \cdot d(B, r)$, and hence,

$$\lambda \cdot stretch(\mathcal{P})(2^\ell - 1) \;\leq\; d(B, r) \;\leq\; \lambda\alpha \cdot stretch(\mathcal{P})(2^{\ell+1} - 1),$$

that is $R(\ell) \leq d(B, r) \leq \alpha \cdot R(\ell + 1)$.

The on-line cost for the replication is is the (amortized) cost of the data tracking update operations, which is bounded by $C_{\mathrm{TRACK}} \cdot D \cdot h$.

As a result of the replication Θ increases by exactly D times the length of the replication path: $\Delta\Theta = D \cdot h$.

Since $h \leq \lambda\alpha \cdot stretch(\mathcal{P})(2^{\ell+1} - 1)$ we obtain

$$\mathrm{Cost}_{\mathrm{DynDFA}} + \Theta \;\leq\; 2\alpha\lambda \cdot D \cdot C_{\mathrm{TRACK}} \cdot stretch(\mathcal{P})(2^{\ell+1} - 1) \tag{4}$$

$$\leq\; 2\beta \cdot (2^{\ell+1} - 1). \tag{5}$$

Let $\mathcal{C}$ be the ℓ-level cluster for which the read counter reached D, when DynDFA decided to replicate to r.

Let the locations of the D requests associated with the replication to r be $r_1, r_2, \ldots r_D = r$. All requests were counted by leader of cluster $\mathcal{C}$, and therefore were issued by processors in $\mathrm{Ker}(\mathcal{C})$.

For $1 \leq j \leq D$, let a_j be the processor nearest to r_j where the intermediate adversary keeps a copy of the file.

Since the intermediate adversary never discards copies, the file copy held nearest to r_j by the adversary at the time of the request, was at least as far from r_j as a_j.

Consider the case that there exists $1 \leq j \leq D$, such that

$$d(a_j, r_j) \leq \frac{1}{4}(2^\ell - 1).$$

Then there exists a covering processor p at the $i(\ell)$-level tree cover at distance of at most $\max\{2(2^{\ell-3} - 1), 0\} \leq (2^\ell - 1)/4$ from a_j.

Therefore,

$$d(p, r_j) \leq d(p_j, a_j) + d(a_j, r_j) \leq \frac{1}{4}(2^\ell - 1) + \frac{1}{4}(2^\ell - 1) = \frac{1}{2}(2^\ell - 1).$$

Since $r_j \in \mathrm{Ker}(\mathcal{C})$, and the $(2^\ell - 1)/2$-neighborhood of p is contained in the $(2^\ell - 1)$-neighborhood of r_j, we have that $p \in \mathrm{ExKer}(\mathcal{C})$.

Prior to the replication the on-line copies were far from the extended-kernel of the cluster:

$$\begin{aligned} d(B, \mathrm{Ker}(\mathcal{C})) &\geq d(B, r) - \max_{q \in \mathrm{Ker}(\mathcal{C})} d(q, r) \\ &> R(\ell) - (2^\ell - 1)\mathit{stretch}(\mathcal{P}) \\ &= (\lambda - 1)\mathit{stretch}(\mathcal{P}) \cdot (2^\ell - 1). \end{aligned}$$

and hence,

$$d(B, \mathrm{Ker}(\mathcal{C})) \geq \max\{(2^\ell - 1), 1\} = \tilde{R}(\ell).$$

After the replication DynDFA holds a file copy at $r \in \mathrm{Ker}(\mathcal{C})$.

It follows that

$$\begin{aligned} \Delta\Psi - \Delta\Lambda &\leq -\lambda^* \cdot D \cdot d_{\tilde{R}(\ell)}(B, \mathrm{Ker}(\mathcal{C})) + \lambda_1^* \cdot D \cdot d_{\tilde{R}(\ell)}(B, \mathrm{Ker}(\mathcal{C})) \\ &\leq (\lambda^* - la_1^*) \cdot D \cdot \tilde{R}(\ell) \\ &\leq -2\beta \cdot (2^{\ell+1} - 1) \end{aligned}$$

Combining with inequality 4 we conclude that

$$\Delta\Phi = \Delta\Psi - \Delta\Lambda + \Delta\Theta \leq -\mathrm{Cost}_{\mathrm{DynDFA}}.$$

Consider now the case that for every $1 \leq j \leq D$,

$$d(a_j, r_j) \;\; \geq \;\; \tfrac{1}{4}(2^\ell - 1).$$

Then we charge the adversary the cost of $d_{\frac{1}{4}\tilde{R}(\ell)}(a_j, r_j)$. This cost is charged only once in each level, since after the replication, for every ℓ level cluster $\mathcal{C}'$ containing r_j, the distance between an on-line file copy and any processor in $\mathcal{C}'$ is at most

$$diam(\mathcal{C}') + d(r_j, r) \;\; \leq \;\; diam(\mathcal{C}') + diam(\mathcal{C}) \;\; \leq \;\; 2(2^\ell - 1) \cdot stretch(\mathcal{P}),$$

and hence less than $\lambda\alpha \cdot (2^\ell - 1) \cdot stretch(\mathcal{P})$, so that a replication will not be initiated by that cluster's leader during this phase.

It follows that the total cost charged for the adversary associated this way, associated with the request at r_j, is at most

$$\sum_{\ell, \text{s.t.} \frac{1}{4}\tilde{R}(\ell) \leq d(a_j, r_j)} \tfrac{1}{4}\tilde{R}(\ell) \;\; \leq \;\; \tfrac{1}{2} \cdot \tilde{R}(m+1) \;\; \leq \;\; 4 \cdot d(a_j, r_j),$$

where m is the largest such that $\frac{1}{4}\tilde{R}(m) \leq d(a_j, r_j)$.

We therefore define the adversary cost associated with level ℓ to be

$$\begin{aligned} \text{Cost}^\ell_{\text{Adv}} \;\; &= \;\; \sum_{j=1}^{D} d_{\frac{1}{4}\tilde{R}(\ell)}(a_j, r_j) \\ &= \;\; \tfrac{1}{4} D \cdot \tilde{R}(\ell). \end{aligned}$$

Therefore, from inequality 4 we obtain

$$\Delta\Phi + \text{Cost}_{\text{DynDFA}} \;\; \leq \;\; 2\beta \cdot D \cdot \tilde{R}(\ell) \;\; \leq \;\; 8 \cdot \text{Cost}^\ell_{\text{Adv}}.$$

■

We are left with dealing with the on-line cost for serving write requests, and with analyzing the potential change at the end of a phase.

The adversary cost associated with a phase is its cost for all requests issued in the phase and for all configuration changes made by the adversary during the phase.

Lemma 5.13 *The change in Φ due to the events associated with the* reorganization step *at the end of a phase is at most $O(\beta \cdot deg(\mathcal{P}) \cdot \min\{\log n, \log(\delta)\})$ times the adversary cost associated with the phase minus the cost* DynDFA *spent for serving the write requests during the phase and for the reorganization step.*

Proof. The lemma follows from the following claims:

Let $\mathrm{Cost}^1_{\mathrm{DFA}}$ be the cost incurred by DFA for the replication onto m.

Let $\overline{B}'$ denote the tree implied by the on-line replications during the phase including the replication to m.

Claim 5.13.1 The total change in the potential function due to the reorganization step is:

$$\begin{aligned}\Delta\Phi \;&\le\; O(\beta\cdot\min\{\log n,\log(\delta)\})\cdot D\cdot\{T(A\cup\{m\})+d(A,m)\}\\ &\quad -\{C_{\mathrm{TRACK}}D\cdot T(\overline{B}')+\mathrm{Cost}^1_{\mathrm{DFA}}\}.\end{aligned}$$

Let $\mathrm{Cost}^2_{\mathrm{DFA}}$ denote the cost DFA incurred for serving the D write requests of the phase, and $\mathrm{Cost}_{\mathrm{Adv}}$ denote the adversary cost for the same requests plus its replications cost.

Claim 5.13.2 The cost the adversary incurs for the write requests of the phase and for its replications made during the phase, satisfies:

$$\mathrm{Cost}_{\mathrm{Adv}} \;\ge\; D\cdot\frac{1}{6}\{T(A\cup\{m\})+d(A,m)\}.$$

Claim 5.13.3 The cost for DFA for the write requests of the phase satisfies:

$$\mathrm{Cost}_{\mathrm{DFA}} \;\le\; C_{\mathrm{TRACK}}\cdot(D\cdot T(\overline{B}')+\mathrm{Cost}_{\mathrm{Adv}}).$$

■

References

[AADW94] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A Theory of Competitive Analysis for Distributed Algorithms. In *Proc. of the 35th Ann. IEEE Symp. on Foundations of Computer Science*, pages 401-411, October 1994.

[ABF93a] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Competitive distributed file allocation. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 164–173, May 1993.

[ABF93b] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Heat & Dump: Randomized competitive distributed paging. In *Proc. 34rd IEEE Symp. on Foundations of Computer Science*. IEEE, November 1993. to appear.

[ACJ+91] Agarwal, Chaiken, Johnson, Kranz, Kubiatowicz, Kurihara, Lim, Maa, and Nussbaum. The mit-alewife machine: A large-scale distributed memory multiprocessor. Mit/lcs/tm 454, MIT, 1991.

[AK94] S. Albers and H. Koga. New On-line Algorithms for the Page Replication Problem. In *Proceedings of the 4th Scandinavian Workshop on Algorithmic Theory*, Aarhus, Denmark, July 1994.

[AK95] S. Albers and H. Koga. Page Migration with Limited Local Memory Capacity. To appear in *Proc. of the 4th Workshop on Algorithms and Data Structures*, , August 1995.

[AP89] Baruch Awerbuch and David Peleg. On-line tracking of mobile users. Technical Memo TM-410, MIT, Lab. for Computer Science, August 1989.

[AP90] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, 1990.

[AP91] Baruch Awerbuch and David Peleg. Concurrent online tracking of mobile users. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Zurich, Switzerland*, September 1991.

[BFR92] Yair Bartal, Amos Fiat, and Yuval Rabani. Competitive algorithms for distributed data management. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 39–50, 1992.

[BLS87] A. Borodin, N. Linial, and M. Saks. An Optimal On-Line Algorithm for Metrical Task Systems. In *Proc. of the 19th Ann. ACM Symp on Theory of Computing*, pages 373–382, May 1987.

[BS89] D.L. Black and D.D. Sleator. Competitive Algorithms for Replication and Migration Problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[Coop92] Keith D. Cooper. Using compiler technology to drive advanced microprocessors. In *DARPA Software Technology Conference*, pages 42–49, April 1992.

[CLRW93] M. Chrobak, L. Larmore, N. Reingold, and J. Westbrook. Optimal Multiprocessor Migration Algorithms Using Work Functions. In *Proc. of the 4th International Symp. on Algorithms and Computation.* Also *Lecture Notes in Computer Science*, vol. 762, pages 406-415, Hong Kong, 1993, Springer-Verlag.

[D$^+$89] William J. Dally et al. The J-Machine: A fine-grain concurrent computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[DF82] D. Dowdy and D. Foster. Comparative models of the file assignment problem. *Computing Surveys*, 14(2), Jun 1982.

[FKL+88] A. Fiat, R.M. Karp, M. Luby, L.A. McGeoch, D.d. Sleator , and N.E. Young. Competitive Paging Algorithms. Technical Report, Carnegie Mellon University, 1988.

[GS90] B. Gavish and O.R.L. Sheng. Dynamic File Migration in Distributed Computer Systems. In *Communications of the ACM*, 33(2):177-189, 1990.

[John92] Kirk Johnson. The impact of communication locality on large-scale multiprocessor performance. In *19th International Symposium on Computer Architecture*. IEEE, May 1992. To appear.

[Koga93] H. Koga. Randomized On-line Algorithms for the Page Replication Problem. In *Proc. of the 4th International Symp. on Algorithms and Computation*. Also *Lecture Notes in Computer Science*, vol. 762, pages 436-445, Hong Kong, 1993, Springer-Verlag.

[KMRS88] Anna Karlin, Mark Manasse, Larry Rudolpoh, and Daniel Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[Lisk92] Barbara Liskov. Preliminary design of the thor object-oriented database system. In *DARPA Software Technology Conference*, pages 50–62, april 1992.

[LLG+90] L. Lenoski, J. Laundo, K. Gharachorloo, A. Gupta, and J.Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proc. of 17th Intern. Symp. on Computer Architecture*, pages 148–159, 1990.

[LMW91] Michael Lam, and Thomas G.Moher, Paul Wilson. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of ACM SIGPLAN*, jun 1991. Conference on Programming Language Design and Implementation.

[LRWY94] C. Lund, N. Reingold, J. Westbrook, and D. Yan. On-Line Distributed Data Management. In *Proc. of European Symp. on Algorithms*, 1994.

[LW91] Monica S. Lam and Michael E. Wolf. Loop transformation theory and algorithm to maximize parallelism. In *IEEE Transactions on Parallel and Distributed Systems*, 1991.

[LW92] Monica S. Lam and Michael E. Wolf. Compilation techniques to achieve parallelism and locality. In *DARPA Software Technology Conference*, pages 150–158, Los Angeles, CA, april 1992.

[ML88] H.L. Morgan and K.D. Levin. Optimal program and data locations in computer networks. *CACM*, 20(5):124–130, 1988.

[MMS88] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive Algorithms for On-Line Problems. In *Proc. of the 20th Ann. ACM Symp. on Theory of Computing*, pages 322-333, May 1988.

[PZ91] Mark Palmer and Stanley B. Zdonik. Fido:a cache that learns to fetch. In *Proceedings of 17th InternationalConference on Very Large Data Bases,Barcelona*, pages 255–264, sept 1991.

[RD90] John T. Robinson and Murthy V. Devarankonda. Data cache management using frequency-based replacement. In *SIGMETRICS, Boulder, CO*, pages 134–142, may 1990. Published as Performance Evaluation Review 18.

[RS89] P. Raghavan and M. Snir. Memory versus Randomization in On-Line Algorithms. In *16th International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 687–703. Springer-Verlag, July 1989.

[ST85] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM*, 28(2):202–208, 1985.

[Stam84] John W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. In *ACM Transactions on Computer Systems*, volume 2, number 2, pages 155–180, May 1984.

[KELS62] T.Kilburn, D.Edwards, M.Lanigan, and F.Summer. One-level storage system. *IRE Transactions on Electronic Computers*, 2:223–235, 1962.

[West91] J. Westbrook. Randomized Algorithms for Multiprocessor Pa ge Migration. In *Proc. of DIMACS Workshop on On-Line Algorithms.* American Mathematical Society, February, 1991.

[WY93] J. Westbrook. and D.K. Yan. Greedy On-Line Steiner Tree and Generalized Steiner Problems. In *Proc. of the 3rd Workshop in Algorithms and Data Structures*, Also *Lecture Notes in Computer Science*, vol. 709, pages 622-633, Montréal, Canada, 1993, Springer-Verlag.

[Young91] N.E. Young. Competitive Paging as Cache Size Varies. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, January 1991.

[YR91] Wang Yongdong and Lawrence A. Rowe. *Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture.* Clifford91, 1991.

Procedure `Read`(F, r) /* read by processor r */

1. Let $s := \mathrm{Find}_F(r)$ be the processor at which a copy of F is found by TRACK(F).
2. Let $h = d(s, r)$. Let ℓ be largest such that $2^\ell - 1 \leq h/(\lambda\alpha \cdot stretch(\mathcal{P}))$.
3. Let $\mathcal{C} = \mathcal{C}_\ell(r)$. Inform cluster leader $L_\ell(\mathcal{C})$ to increase the reads counter $c_\ell(\mathcal{C}, F)$ by 1.
4. If $c_\ell(\mathcal{C}) = D$, then set $c_\ell(\mathcal{C}) = 0$ and call `Replication`$(F, \mathcal{C})$.

Procedure `Replication`$(F, \mathcal{C})$ /* replicate file F into cluster $\mathcal{C}$ */

1. Generate a request for F to the uniprocessor algorithm $\mathrm{U}_\ell(\mathcal{C})$.
2. If no pageout occurs then
   (a) Let $p \in \mathcal{C}$ be a processor assigned with less than $\frac{1}{\wp}k_p$ files by $L_\ell(\mathcal{C})$.
   (b) Replicate F onto p. Input $\mathrm{Insert}_F(s,p)$ to TRACK(F).
   (c) Add F to the list of files assigned to p by $L_\ell(\mathcal{C})$.
3. Else
   (a) Let G be the file paged out by $\mathrm{U}_\ell(\mathcal{C})$,
   (b) Let p be the processor that G ‘ is assigned to by $L_\ell(\mathcal{C})$.
   (c) Delete G from p. Input $\mathrm{Delete}_G(p)$ to TRACK(G).
   (d) Replicate F onto p. Input $\mathrm{Insert}_F(s,p)$ to TRACK(F).
   (e) Add F to the list of files assigned to p by $L_\ell(\mathcal{C})$,
   (f) Delete G from that list.

Figure 4: Hierarchical Paging Algorithm HP, which belongs to class of read-only paging algorithms RO. Calls read-only uniprocessor paging U and distributed data tracking procedure TRACK(F) an arbitrary file F. The latter procedure is called by feeding to it commands Insert, Delete, and Find.

```
Procedure DP(Read, F, p)                              /* read request for F at processor p */
    call RO(p, F)
        q ← the processor F is read from by RO.
    call DynDFA(Read, F, q)                           /* feed to DynDFA read for F from q */
output value returned by DynDFA(Read, F, q)

Procedure DP(Write, F, p)                             /* write request for F at processor p */
    call RO(p, F)
        q ← the processor F is read from by RO.
    call DynDFA(Write, F, q)                          /* feed to DynDFA write for F from q */

whenever RO replaces G at processor p′ by H:
    call DynDFA(Exclude, G, p)
    call DynDFA(Include, H, p)
```

Figure 5: Distributed paging reduces to read-only distributed paging (procedure RO) and file allocation for dynamic networks (procedure DynDFA).

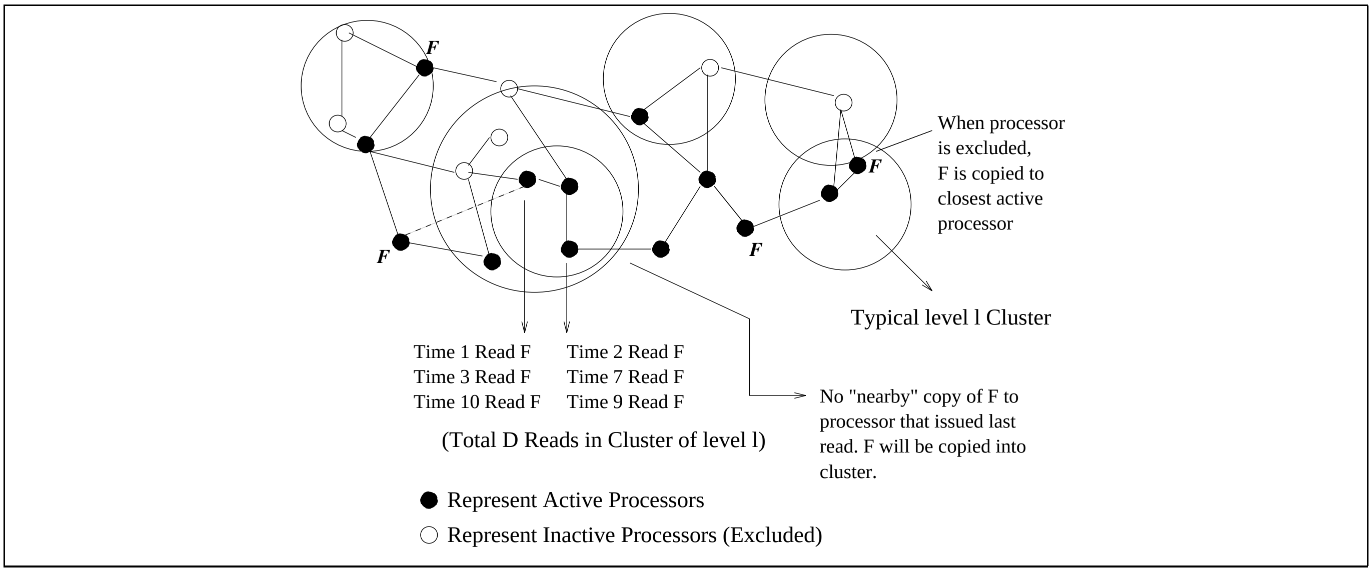

Figure 6: Distributed file allocation for dynamic networks.