Cooperative Computing with Fragmentable and Mergeable Groups

CHRYSSIS GEORGIOU, *Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA. Email: cg2@cse.uconn.edu*

ALEX A. SHVARTSMAN, *Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. Email: alex@theory.lcs.mit.edu*

ABSTRACT: This work considers the problem of performing a set of N tasks on a set of P cooperating message-passing processors ($P \leq N$). The processors use a group communication service (GCS) to coordinate their activity in the setting where dynamic changes in the underlying network topology cause the processor groups to change over time. GCSs have been recognized as effective building blocks for fault-tolerant applications in such settings. Our results explore the efficiency of fault-tolerant cooperative computation using GCSs. The original investigation of this area by Dolev *et al.* [8] focused on competitive lower bounds, non-redundant task allocation schemes and work-efficient algorithms in the presence of fragmentation regroupings. In this work we investigate *work-efficient* and *message-efficient* algorithms for *fragmentation* and *merge* regroupings. We present an algorithm that uses GCSs and implements a coordinator-based strategy. For the analysis of our algorithm we introduce the notion of *view-graphs* that represent the partially-ordered view evolution history witnessed by the processors. For fragmentations *and* merges, the work of the algorithm (defined as the worst case total number of task executions counting multiplicities) is not more than $\min\{N \cdot f + N,\ N \cdot P\}$, and the message complexity is no worse than $4(N \cdot f + N + P \cdot m)$, where f and m denote the number of new groups created by fragmentations and merges respectively. Note that the constants are very small and that, interestingly, while the work efficiency depends on the number of groups f created as the result of fragmentations, work does *not* depend on the number of groups m created as the result of merges.

Keywords: Distributed algorithms, group communication, work, communication, complexity.

1 Introduction

The problem of cooperatively performing a set of tasks in a decentralized setting where the computing medium is subject to failures is one of the fundamental problems in distributed computing. Variations on this problem have been studied in a variety of settings, e.g., in message-passing models [9, 6] and in shared-memory models [15]. This problem was also studied in the setting of processor groups in partitionable networks [8]. In this setting, the computation can take advantage of group communication services [4], and the processors must perform the tasks and must learn the results of the tasks efficiently, despite the dynamically changing group memberships.

Group communication services (GCS) can be used as effective building blocks for constructing fault-tolerant distributed applications. These services enable the application components at different processors to operate collectively as a group, using the service to multicast messages. The basis of a group communication service is a *group membership service*. Each processor, at each time, has a unique *view* of the membership of the group. The view includes a list of the processors that are members of the group. Views can change and may become different at different processors. There is a substantial amount of research dealing with specification and implementation of GCSs and group-oriented applications, e.g., [1, 2, 14, 7, 10, 17, 21, 23], and verification of GCSs and group-oriented systems, e.g., [5, 16, 11].

When developing group-oriented, and especially partition-aware applications, it is also important to understand the effectiveness of group communication services [25] and the efficiency benefits that can be expected when using group communication services [8]. One of the features of GCSs is their group management facilities that map a variety of failures in the underlying computing medium to changes in group memberships. Faulty communication links can partition the system into several connected components. Failures and recoveries trigger group membership activity that aims to establish a group for every connected component. An adversary that causes frequent and arbitrary failures may prevent applications from making steady computational progress. Thus, it is interesting to study restricted, yet realistic, models of adversaries for which efficient specific algorithms can be developed with the help of common group communication services. Studying the problem of performing a set of tasks on a set of processors in the group-oriented setting provides a convenient and powerful abstraction for understanding the efficiency of cooperative computation. A work-efficient algorithm is presented for this problem by Dolev *et al.* in [8], along with a lower bound and a scheduling strategy that minimizes redundant work. That algorithm is tolerant of arbitrary sequences of group fragmentations. In this work we present the study of algorithms that are work-efficient and message-efficient, and that are able to deal with more general changes in group memberships.

Following [8], we investigate an approach whose goal is to utilize the resources of *every component* of the system during the entire computation. The problem we consider has the following setting: *a set of N independent and idempotent tasks must be performed by P processors in a distributed system, where each processor must learn all results*. Group communication is used to coordinate the execution of the tasks. Our distributed system model, in addition to the processors and the network, includes a set of input/output ports accessible to the processors. In this model we enable any client of the required computation to query any processor for the results. This makes it mandatory, even for isolated processors, to be able to provide the results of the computation regardless of whether any other processors may already have the results. Thus, it is not sufficient to know that each of the tasks have been performed somewhere. It is also necessary for each processor to learn the results. We refer to this problem as the OMNI-DO problem.

Note that any algorithm that solves the problem in a way where in any group the processors perform no more than $\Theta(N)$ tasks (counting multiplicities), will have work complexity of $O(r \cdot N)$, where r is the total number of new views installed. This makes it not very interesting to study the problem for adversaries that impose arbitrary view changes. Our major goal is to develop *precise* upper bounds that describe the work and messaging efficiency of solving OMNI-DO as functions of the number of tasks N, the number of processors P, and the numbers of distinct group views of *specific* types (fragmentations and merges in this work) installed by the group membership services.

We present an algorithm for the OMNI-DO problem for N tasks and P message-passing processors ($P \leq N$) that are interconnected by a network, which is subject to dynamic group fragmentations and merges. We assume a group communication service that provides group management and view-oriented messaging service (Section 2.2). The main complexity result is for the adversary that is restricted to causing fragmentations of groups and merges of groups. This extends the results in [8], which consider only the fragmentation adversary. Our analysis for the fragmentation-and-merge adversary yields analysis for the fragmentations-only adversary as a corollary.

For the fragmentation-and-merge adversary, we distinguish between the views that are installed as the result of fragmentations and the views installed as the result of merges. If r is the total number of views installed, then for the fragmentation-and-merge adversary we have that $r = f + m$, where f is the number of views due to fragmentations and m is the number of views due to merges. It is also not difficult to see that $m < f$ when all processors initially start in a single group.

The fragmentation-and-merge adversary is more powerful than the fragmentation adversary of [8] and it can cause the degradation of computation efficiency, e.g., by merging groups it can increase the message cost (as implied by our results) and it can cause more numerous fragmentations (in the fragmentations-only model there can be at most $P - 1$ fragmentations). Intuitively it is reasonable to expect that while merges may not degrade work efficiency, they may increase the messaging due to the additional coordination overhead. Our analysis confirms this intuition.

We now summarize our results.

- We present a new algorithm, called algorithm *AX*, that solves the OMNI-DO problem and we analyze it for the fragmentation-and-merge adversary. The algorithm employs a coordinator-based approach and relies on the underlying group communication service. The algorithm is specified in Section 4.
- We introduce the notion of *view-graphs* that represent the partially-ordered view evolution history collectively witnessed by the processors (Section 3). We show that these digraphs are acyclic for the fragmentation-and-merge adversary and we use these view-graphs in the complexity analysis of the algorithm. We believe that view-graphs have the potential of serving as a general tool for studying cooperative computing with group communication services.
- For any pattern of fragmentations and merges, the work W of the algorithm is no more than $\min\{N \cdot f + N,\ N \cdot P\}$, and the message complexity M is no worse than $4(N \cdot f + N + P \cdot m)$. Note that $f \leq r$ and here it is significant that we are expressing the upper bounds using explicit constants instead of the big-oh notation. Both complexity results depend on f, but only the message complexity depends

on m. These facts substantiate the intuition that merges lead to a more efficient computation, but require additional coordination. This analysis is presented in Sections 5.1 and 5.2.

- For any pattern of fragmentations (i.e., when $m = 0$) our algorithm achieves work complexity of $O(\min\{N \cdot f + N, N \cdot P\})$. This result is essentially the same as the result in [8]. However, our algorithm achieves substantially better *message complexity* $O(N \cdot f + N)$ as compared to the at least quadratic message complexity of the algorithm in [8]. Message optimization was outside of the scope of [8], yet this improvement was one of our goals. The improvement is largely due to our use of the coordinator-based strategy. These results are in Section 5.3.

Note that it is not difficult to see that if $f \geq P$, then it is always possible to produce an execution such that $W = \Omega(N \cdot P)$, and if $f < P$, then it is possible to produce an execution such that $W = \Omega(N \cdot f)$. Thus, $W = \Omega(\min\{N \cdot f,\ N \cdot P\})$ is a lower bound for OMNI-DO. This makes our algorithm work-optimal with respect to the adversaries we consider. Considering optimality for the message complexity is less interesting, since the problem can be solved without any communication (cf. [20]).

Related work. The problem of efficiently performing a set of tasks using a network of processors in the setting where the network is subject to dynamic changes was considered by Dolev, Segala and Shvartsman [8]. For the N-processor, N-task problem defined in that work, it was shown that for dynamic changes the termination time of any on-line task algorithm can be greater than the termination time of an off-line algorithm by a factor linear in N. An algorithm was also presented in [8] that for arbitrary fragmentations has work $O(N \cdot f' + N)$, where f' is the increase in the number of groups due to fragmentations. In comparing our result with the result in [8], we note that our definition of f is slightly different from the definition of fragmentation failures f' in [8]. In order to compare our complexity results with those in [8], we show in this paper that for any pattern of fragmentations allowed by [8] we have $f' < f < 2f'$. In [8] the work is counted in terms of the rounds executed by the processors. In our analysis we count only the number of task executions (including redundancies). However in our algorithm, for as long as any tasks remain undone in a given group, the processors perform the tasks in rounds, except for the last round. Therefore the difference in work complexity for these two algorithms is at most $f \cdot N$. Thus the different definitions of f and f' and of work can be subsumed in the big-oh analysis without substantial variation in the constants.

Group communication services (GCS) have become important as building blocks for fault-tolerant distributed systems. Such services enable processors located in a fault-prone network to operate collectively as a group, using the services to multicast messages to group members. Examples of GCS include Isis [2], Transis [7], Totem [21], Newtop [10], Relacs [1], Horus [23] and Ensemble [14]. Examples of recent work dealing with primary groups are [5, 17]. An example of an application using a GCS for load balancing is by Fekete, Khazan and Lynch [16]. To evaluate the effectiveness of partitionable GCSs, Sussman and Marzulo [25] proposed the measure (*cushion*) precipitated by a simple partition-aware application.

Our definition of work follows that of Dwork, Halpern and Waarts [9]. Our fragmentation model creates a setting, within each fragment, that is similar to the setting in which the network does not fragment but the processors are subject to crash failures. Performing a set of tasks in such settings is the subject of several works [3, 6, 9, 12], however the analysis is quite different when work in all fragments has to be considered.

Our distributed problem has an analogous counterpart in the shared-memory model of computation, called the *collect* problem. The collect problem was originally abstracted by Saks, Shavit and Woll [24] (it also appears in Shavit's Ph.D. thesis). Although the algorithmic techniques are different, the goal of having all processors to learn a set of values is similar.

The rest of the paper is structured as follows. In Section 2 we describe models, assumptions and complexity measures. In Section 3 we introduce and define view graphs and the adversary models. In Section 4 we describe Algorithm *AX* and in Section 5 we give its complexity analysis. We conclude in Section 6 with a discussion.

A preliminary version of this paper appeared as [13].

2 Definition and Models

We begin by presenting the system model, the group communication service properties, and work and communication complexity measures.

2.1 The System Model and the OMNI-DO *Problem*

The distributed system consists of P processors connected by communication links. Each processor has a unique identifier from the set $\mathcal{P} = \{1, 2, \ldots, P\}$.

We define a *task* to be any computation that can be performed by a single processor in constant time. We assume that the tasks are independent and idempotent. Our distributed system is charged with the responsibility of performing a set of N tasks that are initially known to all processors. Each task has a unique identifier from the set $\mathcal{T}$.

To require that all processors acquire the results of all tasks, our system also includes a set of input/output ports. These ports are only used by the clients of the system to query individual processors for computation results. We do not make any failure assumptions about the input/output ports, in particular, our algorithm does not depend on the failure status of these ports, or the requests from them.

DEFINITION 2.1
The problem of performing a set of N independent tasks on a set of P message passing processors, where each processor must learn the results of all N tasks, is called the OMNI-DO problem.

The algorithm specification in this paper is done in terms of I/O automata of Lynch and Tuttle [18, 19]. Each automaton models a state machine with states and transitions between states, where actions are associated with sets of state transitions. There are input, output and internal actions. A particular action is enabled if the preconditions of that action are satisfied. The statements given as effects are executed as a program started in the existing state and atomically producing the next state as the result of the transition.

An *execution* α of an I/O automaton Aut is a finite or infinite sequence of alternating states and actions (events) of Aut starting with the initial state, i.e., $\alpha = s_0, e_1, s_1, e_2, \ldots$, where s_i's are states (s_0 is the initial state) and e_i's are actions (events). We denote by *execs*(Aut) the set of all executions in Aut.

We next state our assumptions about the group communication services and define the work and message complexity measures.

2.2 Group Communication Service

We assume a group communication service (GCS) with certain properties. The assumptions are basic, and they are provided by several group communication systems and specifications [26]. The service maintains group membership information and it is used to communicate information concerning the executed tasks within each group. The GCS provides the following primitives:

- NEWVIEW$(v)_p$: informs processor p of a new view $v = \langle id, set \rangle$, where id is the identifier of the view and set is the set of processor identifiers in the group. When a NEWVIEW$(v)_p$ primitive is invoked, we say that processor p *installs* view v.
- GPMSND(*message*)$_p$: processor p multicasts a message to the group members.
- GPMRCV(*message*)$_p$: processor p receives multicasts from other processors.
- GP1SND(*message,destination*)$_p$: processor p unicasts a message to another member of the current group.
- GP1RCV(*message*)$_p$: processor p receives unicasts from another processor.

To distinguish between the messages sent in different send events, we assume that each message sent by the application is tagged with a unique message identifier.
We assume the following safety properties on any execution α of an algorithm that uses GCSs:

1. A processor is always a member of its view ([26] Prop. 3.1). If NEWVIEW$(v)_p$ occurs in α then $p \in v.set$.
2. The view identifiers of the views that each processor installs are monotonically increasing ([26] Prop. 3.2). If event NEWVIEW$(v_1)_p$ occurs in α before event NEWVIEW$(v_2)_p$, then $v_1.id < v_2.id$. This property implies that:
   (a) A processor does not install the same view twice.
   (b) If two processors install the same two views, they install these views in the same order.
3. For every receive event, there exists a preceding send event of the same message ([26] Prop. 4.1). If GPMRCV$(m)_p$ (GP1RCV$(m)_p$) occurs in α, then there exists GPMSND$(m)_q$ (GP1SND$(m, p)_q$) earlier in execution α.
4. Messages are not duplicated ([26] Prop. 4.2). If GPMRCV$(m_1)_p$ (GP1RCV$(m_1)_p$) and GPMRCV$(m_2)_p$ (GP1RCV$(m_2)_p$) occur in α, then $m_1 \neq m_2$.

5. A message is delivered in the same view it was sent in ([26] Prop. 4.3). If processor p receives message m in view v_1 and processor q (it is possible that $p = q$) sends m in view v_2, then $v_1 = v_2$.
6. In the initial state s_0, all processors are in the initial view v_0, such that $v_0.set = \mathcal{P}$ ([26] Prop. 3.3 with [11, 22]).

We assume the following additional liveness properties on any execution α of an algorithm that uses GCSs (cf. [26] Section 10):

7. If a processor p sends a message m in the view v, then for each processor q in $v.set$, either q delivers m in v, or p installs another view.
8. If a new view event occurs at any processor p in view v, then a view change will eventually occur at all processors in $v.set - \{p\}$.

2.3 Regrouping-Numbers and Measures of Efficiency

In this section we define regrouping-numbers and complexity measures. We define the *regrouping-number* r of an execution to be the number of NEWVIEW events with distinct view identifiers. (Note that if the same view is installed at multiple processors, this counts for a single regrouping.)

DEFINITION 2.2
Given an execution α, we define the *regrouping-number* r_α as:

$$r_\alpha = |\{v : \text{NEWVIEW}(v)_p \text{ occurs in } \alpha\}|.$$

When it is clear from the context, we use r instead of r_α to denote the regrouping-number of execution α.

We define *adversary models*, in the context of a specific algorithm, in terms of the collections of executions in the presence of an adversary.

DEFINITION 2.3
For an algorithm A, let $\mathcal{F}_R(A)$ be the adversary model that includes all possible executions of A, i.e., $\mathcal{F}_R(A) = execs(A)$, and let $\mathcal{F}_\emptyset(A)$ be the adversary model that does not cause any NEWVIEW events, i.e., $\mathcal{F}_\emptyset(A) = \{\alpha : \alpha \in execs(A) \wedge r_\alpha = 0\}$.

When it is clear from the context, we use $\mathcal{F}_\emptyset$ instead of $\mathcal{F}_\emptyset(A)$ and $\mathcal{F}_R$ instead of $\mathcal{F}_R(A)$. It is easy to see that $\mathcal{F}_\emptyset \subseteq \mathcal{F}_R$. Let $\mathcal{F}$ be some adversary model such that $\mathcal{F}_\emptyset \subseteq \mathcal{F} \subseteq \mathcal{F}_R$. In the following definitions we formalize the measures of work and message complexity for the specific $\mathcal{F}$. Our definition of work follows that of Dwork, Halpern and Waarts [9].

DEFINITION 2.4
The *work* $W_\alpha(N, P)$ of an execution α of algorithm A in the adversary model $\mathcal{F}$, is defined to be $\sum_{i \in \mathcal{P}} W_\alpha^i$, where W_α^i is the number of tasks performed by processor i. The *work complexity* $W_\mathcal{F}(N, P, r)$ is defined as:

$$W_\mathcal{F}(N, P, r) = \max_{\alpha \in \mathcal{F}, r_\alpha \leq r} \{W_\alpha(N, P)\}.$$

DEFINITION 2.5
The *message cost* $M_\alpha(N, P)$ of an execution α of algorithm A in the adversary model $\mathcal{F}$, is defined to be $\sum_{i \in \mathcal{P}} M^i_\alpha$, where M^i_α is the number of messages sent by processor i. The *message complexity* $M_\mathcal{F}(N, P, r)$ is defined as:

$$M_\mathcal{F}(N, P, r) = \max_{\alpha \in \mathcal{F}, r_\alpha \leq r} \{M_\alpha(N, P)\}.$$

3 View-Graphs and Specific Adversary Models

This section introduces *view-graphs* that represent view changes at processors in executions and that are used to analyze properties of executions. View-graphs are directed graphs (digraphs) that are defined by the states and by the NEWVIEW events of executions of algorithms that use group communication services. Representing view changes as digraphs enables us to use common graph analysis techniques to formally reason about the properties of executions. In this paper we deal with adversary models that cause group fragmentations and merges. Although the meaning of such reconfigurations seems very intuitive, it is necessary to carefully define them to enable formal reasoning. Our view-graph approach to the analysis of executions is general, and we believe it can be used to study other properties of group communication services and algorithms for different adversary models.

3.1 Executions and View-Graphs

Consider an algorithm A that uses a group communication service (GCS). We modify algorithm A by introducing, for each processor i, the history variable cv_i that keeps track of the current view at i as follows: In the initial state, we set cv_i to be v_0, the distinguished initial view for all processors $i \in \mathcal{P}$. In the effects of the NEWVIEW$(v)_i$ action for processor i, we include the assignment $cv_i := v$. In the rest of the paper, we assume that algorithms are modified to include such history variables. We now define *view-graphs* by specifying how a view-graph is induced by an execution of an algorithm.

DEFINITION 3.1
Given an execution α of algorithm A, the *view-graph* $\Gamma_\alpha = \langle V, E, L\rangle$ is defined to be the labeled directed graph as follows:

1. Let V_α be the set of all views v that occur in NEWVIEW$(v)_i$ events in α. The set V of nodes of Γ_α is the set $V_\alpha \cup \{v_0\}$. We call v_0 the initial node of Γ_α.
2. The set of edges E of Γ_α is a subset of $V \times V$ determined as follows. For each NEWVIEW$(v)_i$ event in α that occurs in state s, the edge $(s.cv_i, v)$ is in E.
3. The edges in E are labeled by $L : E \to 2^\mathcal{P}$, such that $L(u, v) = \{i :$ NEWVIEW$(v)_i$ occurs in state s in α such that $s.cv_i = u\}$.

Observe that the definition ensures that all edges in E of Γ_α are labeled.

EXAMPLE 3.2
Consider the following execution α (we omit all events other than NEWVIEW and any states that do not precede NEWVIEW events):

$$\alpha = s_0, \text{NEWVIEW}(v_1)_{p_1}, \ldots, s_1, \text{NEWVIEW}(v_2)_{p_2}, \ldots, s_2, \text{NEWVIEW}(v_3)_{p_4}, \ldots,$$
$$s_3, \text{NEWVIEW}(v_4)_{p_1}, \ldots, s_4, \text{NEWVIEW}(v_1)_{p_3}, \ldots, s_5, \text{NEWVIEW}(v_4)_{p_2}, \ldots,$$
$$s_6, \text{NEWVIEW}(v_4)_{p_3}, \ldots,$$

where $v_1.set = \{p_1, p_3\}$, $v_2.set = \{p_2\}$, $v_3.set = \{p_4\}$ and $v_4.set = \{p_1, p_2, p_3\}$. Additionally, $v_0.set = \mathcal{P} = \{p_1, p_2, p_3, p_4\}$.

The view-graph $\Gamma_\alpha = \langle V, E, L\rangle$ is given in Figure 1. The initial node of Γ_α is v_0. The set of nodes of V of Γ_α is $V = V_\alpha \cup \{v_0\} = \{v_0, v_1, v_2, v_3, v_4\}$. The set of edges E of Γ_α is $E = \{(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_4), (v_2, v_4)\}$, since for each of these (v_j, v_k) the event $\text{NEWVIEW}(v_k)_p$ occurs in state s_ℓ where $s_\ell.cv_p = v_j$ for some certain p (by the definition of the history variable). The labels of the edges are $L(v_0, v_1) = \{p_1, p_3\}$, $L(v_0, v_2) = \{p_2\}$, $L(v_0, v_3) = \{p_4\}$, $L(v_1, v_4) = \{p_1, p_3\}$ and $L(v_2, v_4) = \{p_2\}$, since for each $p_i \in L(v_j, v_k)$ the event $\text{NEWVIEW}(v_k)_i$ occurs in state s_ℓ where $s_\ell.cv_{p_i} = v_j$.

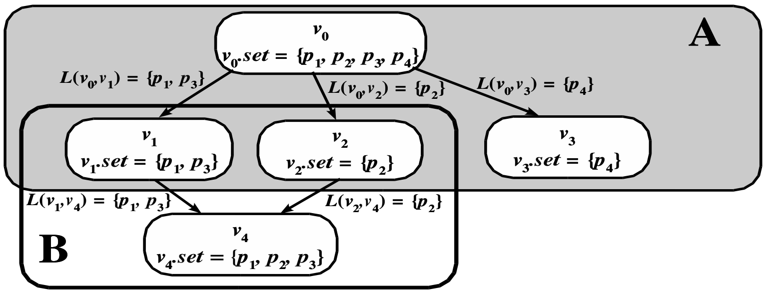

FIG. 1. Example of a view-graph

Given a graph S and a node v of S, we define *indegree*(v, S) (*outdegree*(v, S)) to be the indegree (outdegree) of v in S.

LEMMA 3.3
For any execution α, *indegree*$(v_0, \Gamma_\alpha) = 0$.

PROOF. In the initial state s_0, $s_0.cv$ is defined to be v_0 for all processors in $\mathcal{P}$ and $v_0.set = \mathcal{P}$. Assume that *indegree*$(v_0, \Gamma_\alpha) > 0$. By the construction of view-graphs, this implies that some processor $i \in \mathcal{P}$ installs v_0 a second time. But this contradicts the property 2(a) of GCS. ∎

LEMMA 3.4
Let α be an execution and $\Gamma_\alpha|_i$ be the projection of Γ_α on the edges whose label includes i, for some $i \in \mathcal{P}$. $\Gamma_\alpha|_i$ is an elementary path and v_0 is the path's source node.

PROOF. Let execution α be $s_0, e_1, s_1, e_2, \ldots$. Let $\alpha^{(k)}$ be the prefix of α up to the k^{th} state. i.e., $\alpha^{(k)} = s_0, e_1, s_1, e_2, \ldots, s_k$. Let Γ^k_α be the view-graph that is induced by $\alpha^{(k)}$. Then define $\Gamma^k_\alpha|_i$ to be the projection of Γ^k_α on the edges whose label includes i, for some $i \in \mathcal{P}$. For an elementary path π, we define $\pi.sink$ to be its sink node.

We prove by induction on k that $\Gamma_\alpha^k|_i$ is an elementary path, that $\Gamma_\alpha^k|_i.sink = s_k.cv_i$ and that v_0 is the path's source node.

Basis: $k = 0$. $\Gamma_\alpha^0|_i$ has only one vertex, v_0, and no edges ($\alpha^{(0)} = s_0$). Thus, $\Gamma_\alpha^0|_i.sink = s_0.cv_i = v_0$ and v_0 is the source node of this path.

Inductive Hypothesis: Assume that $\forall n \leq k$, $\Gamma_\alpha^n|_i$ is an elementary path, that $\Gamma_\alpha^n|_i.sink = s_n.cv_i$ and that v_0 is the path's source node.

Inductive Step: $n = k + 1$. For state s_{k+1} we consider two cases:

Case 1: If event e_{k+1} is not a NEWVIEW event involving processor i, then $\Gamma_\alpha^{k+1}|_i = \Gamma_\alpha^k|_i$. Thus, by inductive hypothesis, $\Gamma_\alpha^{k+1}|_i$ is an elementary path and v_0 is its source node. From state s_k to state s_{k+1}, processor i did not witness any new view. By the definition of the history variable, $s_{k+1}.cv_i = s_k.cv_i$. Thus, $\Gamma_\alpha^{k+1}|_i.sink = s_k.cv_i = s_{k+1}.cv_i$.

Case 2: If event e_{k+1} is a NEWVIEW$(v)_i$ event that involves processor i, then by the construction of the view-graph, $(s_k.cv_i, v)$ is a new edge from node $s_k.cv_i$ to node v. By inductive hypothesis, $\Gamma_\alpha^k|_i.sink = s_k.cv_i$. Since our GCS does not allow the same view to be installed twice (property 2(a)), $v \neq u$ for all $u \in \Gamma_\alpha^k|_i$. Thus, $\Gamma_\alpha^{k+1}|_i$ is also an elementary path, with v_0 its source node and $\Gamma_\alpha^{k+1}|_i.sink = v$. From state s_k to state s_{k+1}, processor i installs the new view v. By the definition of the history variable, $s_{k+1}.cv_i = v$. Thus, $\Gamma_\alpha^{k+1}|_i.sink = s_{k+1}.cv_i$. This completes the proof. ■

THEOREM 3.5
Any view-graph Γ_α, induced by any execution α of algorithm A is a connected graph.

PROOF. The result follows from Definition 3.1(2), from the observation that all edges of the view-graph are labeled and from Lemma 3.4 ■

DEFINITION 3.6
For a view-graph $\Gamma_\alpha = \langle V, E, L\rangle$, a *fragmentation subgraph* is a connected labeled subgraph $S = \langle V_S, E_S, L_S\rangle$ of Γ_α such that:

1. S contains a unique node v such that $indegree(v, S) = 0$; v is called the *fragmentation node* of S.
2. $V_S = \{v\} \cup V'_S$, where V'_S is defined to be $\{w : (v, w) \in E\}$.
3. $E_S = \{(v, w) : w \in V'_S\}$.
4. L_S is the restriction of L on E_S.
5. $\bigcup_{w \in V'_S}(w.set) = v.set$.
6. $\forall u, w \in V'_S$ such that $u \neq w,\ u.set \cap w.set = \emptyset$.
7. $\forall w \in V'_S$, $L_S(v, w) = w.set$.

In the analysis of algorithms, we are going to be referring to all NEWVIEW events that collectively induce a fragmentation subgraph for a fragmentation node v as a *fragmentation*.

EXAMPLE 3.7
The shaded area A in Figure 1 shows the fragmentation subgraph $S = \langle V_S, E_S, L_S\rangle$ of Γ_α from Example 3.2. Here $V_S = \{v_0, v_1, v_2, v_3\},\ E_S = \{(v_0, v_1), (v_0, v_2),$

$(v_0, v_3)\}$ and the labels are the labels of Γ_α restricted on E_S. We can confirm that S is a fragmentation subgraph by examining the individual items of Definition 3.6.

DEFINITION 3.8
For a view-graph $\Gamma_\alpha = \langle V, E, L\rangle$, a *merge subgraph* is a connected labeled subgraph $S = \langle V_S, E_S, L_S\rangle$ of Γ_α such that:

1. S contains a unique node v such that *outdegree*$(v, S) = 0$ and *indegree*$(v, S) > 1$; v is called the *merge node* of S.
2. $V_S = \{v\} \cup V'_S$, where V'_S is defined to be $\{w : (w, v) \in E\}$.
3. $E_S = \{(w, v) : w \in V'_S\}$.
4. L_S is the restriction of L on E_S.
5. $\bigcup_{w\in V'_S}(w.set) = v.set.$
6. $\forall u, w \in V'_S$ such that $u \neq w,\ u.set \cap w.set = \emptyset.$
7. $\bigcup_{w\in V'_S} L_S(w, v) = v.set.$

A regrouping of a group g_1 to a group g_2 such that $g_1.set = g_2.set$ can be represented either as a fragmentation subgraph or as a merge subgraph. In this paper we choose to represent it as a fragmentation subgraph by requiring that *indegree*$(v, S) > 1$ for any merge node v.

In the analysis of algorithms, we are going to be referring to all NEWVIEW events that collectively induce a merge subgraph for a merge node v as a *merge*.

EXAMPLE 3.9
The area B in Figure 1 of Example 3.2 shows the merge subgraph $S = \langle V_S, E_S, L_S\rangle$ of Γ_α, where $V_S = \{v_1, v_2, v_3, v_4\}$, $E_S = \{(v_1, v_4), (v_2, v_4)\}$ and the labels are the labels of Γ_α restricted on E_S. We can verify this by examining all conditions of Definition 3.8.

DEFINITION 3.10
Given a view-graph Γ_α we define:

(a) *frag*(Γ_α) to be the set of all the distinct fragmentation nodes in Γ_α,
(b) *merg*(Γ_α) to be the set of all the distinct merge nodes in Γ_α.

DEFINITION 3.11
Given a view-graph Γ_α:

(a) if all of its non-terminal nodes are in *frag*(Γ_α), then Γ_α is called a *fragmentation view-graph*.
(b) if each of its non-terminal nodes is either in *frag*(Γ_α), or it is an immediate ancestor of a node which is in *merg*(Γ_α), then Γ_α is called an *fm view-graph*.

For Γ_α in the example in Figure 1 we have $v_0 \in$ *frag*(Γ_α) by Definition 3.10(a). Also, $v_4 \in merg(\Gamma_\alpha)$ per Definition 3.10(b); additionally, the nodes v_1 and v_2 are immediate ancestors of $v_4 \in merg(\Gamma_\alpha)$. By Definition 3.11(b), Γ_α is an fm view-graph. Observe that Γ_α is a DAG. This is true for all view-graphs:

THEOREM 3.12
Any view-graph $\Gamma_\alpha = \langle V, E, L\rangle$ is a Directed Acyclic Graph (DAG).

PROOF. Assume that Γ_α is not a DAG. Thus, it contains at least one cycle. Let $((v_1, v_2)(v_2, v_3)\ldots(v_k, v_1))$ be an elementary cycle of Γ_α. By the construction of view-graphs (Definition 3.1(3)) and by the monotonicity property (property 2) of GCS, $v_i.id < v_{i+1}.id$ for $1 \le i \le k$ and $v_k.id < v_1.id$. But, by the transitivity of "<", $v_1.id < v_k.id$, a contradiction. ∎

COROLLARY 3.13
Any fm view graph is a DAG and any fragmentation view-graph is a rooted tree.

In the complexity analysis we use the following fact.

FACT 3.14
In any (non-empty) DAG, there is at least one vertex, such that all of its descendants have outdegree 0.

3.2 Adversary Models

Let A be an algorithm that uses GCS, as presented in Section 2.2. We now define two adversary models that are more restrictive than $\mathcal{F}_R(A)$, but less restrictive than $\mathcal{F}_\emptyset(A)$.

DEFINITION 3.15
For any algorithm A the *fragmentation adversary* $\mathcal{F}_F(A)$ is the set of all executions of A, such that each execution induces a fragmentation view-graph. The *fragmentation-and-merge adversary* $\mathcal{F}_{FM}(A)$ is the set of all executions of A, such that each execution induces an fm view-graph.

It is easy to see that $\mathcal{F}_\emptyset(A) \subseteq \mathcal{F}_F(A) \subseteq \mathcal{F}_{FM}(A) \subseteq \mathcal{F}_R(A)$.

DEFINITION 3.16
Given an execution α of algorithm A, and $\Gamma_\alpha = \langle V, E, L\rangle$, we define:

1. the *fragmentation-number* $f_\alpha = |\{w : \text{NEWVIEW}(w)_p \text{ occurs in } \alpha \;\wedge\; (v, w) \in E \;\wedge\; v \in \mathit{frag}(\Gamma_\alpha)\}|$,
2. the *merge-number* $m_\alpha = |\{v : \text{NEWVIEW}(v)_p \text{ occurs in } \alpha \;\wedge\; v \in \mathit{merg}(\Gamma_\alpha)\}|$.

Note that for an algorithm A and for an execution $\alpha \in \mathcal{F}_{FM}(A)$, by Definitions 2.2 and 3.16, $r_\alpha = f_\alpha + m_\alpha$. Also, by Definitions 3.10 and 3.16, $f_\alpha > m_\alpha$. Observe that in the adversary model $\mathcal{F}_F,\ r_\alpha = f_\alpha$ and $m_\alpha = 0$.

4 Algorithm *AX*

We now present the algorithm, called algorithm *AX*, that deals with *regroupings* and that relies on a GCS as specified in Section 2.2. The analysis of the algorithm is in Section 5.

Algorithm *AX* uses a coordinator approach within each group view. The high level idea of the algorithm is that each processor performs (remaining) tasks according to a load balancing rule, and a processor completes its computation when it learns the results of all the tasks.

Task allocation. The set T of the initial tasks is known to all processors. During the execution each processor i maintains a local set D of tasks already done, a local set R of the corresponding results, and the set G of processors in the current group. (The set D may be an underestimate of the set of tasks done globally.) The processors allocate tasks based on the shared knowledge of the processors in G about the tasks done. For a processor i, let $rank(i,G)$ be the rank of i in G when processor identifiers are sorted in ascending order. Let U be the tasks in $T - D$. For a task u in U, let $rank(u,U)$ be the rank of u in U when task identifiers are sorted in ascending order. Our *load balancing rule* for each processor i in G is that:

- if $rank(i,G) \leq |U|$, then processor i performs task u such that $rank(u,U) = rank(i,G)$;
- if $rank(i,G) > |U|$, then processor i does nothing.

Algorithm structure. The algorithm code is given in Figure 2 using I/O automata notation [19]. The algorithm uses the group communication service to structure its computation in terms of *rounds* numbered sequentially within each group view.

Initially all processors are members of the distinguished initial view v_0, such that $v_0.set = \mathcal{P}$. Rounds numbered 1 correspond to the initial round either in the original group or in a new group upon a regrouping as notified via the NEWVIEW event. If a regrouping occurs, the processor receives the new set of members from the group membership service and starts the first round of this view (NEWVIEW action). At the beginning of each round, denoted by a round number Rnd, processor i knows G, the local set D of tasks already done, and the set R of the results. Since all processors know G, they "elect" the group coordinator to be the processor which has the highest processor id (no communication is required since the coordinator is uniquely identified). In each round each processor reports D and R to the coordinator of G (GP1SND action). The coordinator receives and collates these reports (GP1RCV action) and sends the result to the group members (GPMSND action). Upon the receipt of the message from the coordinator, processors update their D and R, and perform work according to the load balancing rule (GPMRCV action).

For generality, we assume that the messages may be delivered by the GCS out of order. The set of messages within the current view is saved in the local variable A. The saved messages are also used to determine when all messages for a given round have been received. Processing continues until each member of G knows all results (the processors enter the *sleep* stage). When requests for computation results arrive from a port q (REQUEST action), each processor keeps track of this in a local variable *requests*, and, when all results are known, sends the results to the port (REPORT action).

The variables cv and MSG are *history variables* that do not affect the algorithm, but play a role in its analysis.

Correctness: We now show the safety of algorithm AX. We first show that no pro-

Data types and identifiers:

$\mathcal{T}$: tasks
$\mathcal{R}$: results
$Result : \mathcal{T} \rightarrow \mathcal{R}$
$\mathcal{M}es$: messages
$\mathcal{P}$: processor ids
$\mathcal{G}$: group ids
$views = \mathcal{G} \times 2^{\mathcal{P}}$: views, selectors id and set
$\mathcal{IO}$: input/output ports

$m \in \mathcal{M}es$
$i, j \in \mathcal{P}$
$v \in views$
$H \in 2^{\mathcal{T}}$
$Q \in 2^{\mathcal{R}}$
$round \in \mathbf{N}$
$results \in 2^{\mathcal{R}}$
$q \in \mathcal{IO}$

States:

$T \in 2^{\mathcal{T}}$, the set of $N = |T|$ tasks
$D \in 2^{\mathcal{T}}$, the set of done tasks, initially $\emptyset$
$R \in 2^{\mathcal{R}}$, the set of known results, initially $\emptyset$
$G \in 2^{\mathcal{P}}$, current members, init. $v_0.set = \mathcal{P}$
$A \in 2^{\mathcal{M}es}$, messages since last NEWVIEW, initially $\emptyset$
$Rnd \in \mathbf{N}$, round number, initially 1
$requests \in 2^{\mathcal{IO}}$, set of ports, initially $\emptyset$
$Phase \in \{send, receive, sleep, mcast, mrecv\}$, initially $send$

Derived variables:

$U = T - D$, the set of remaining tasks
$Coordinator(i)$: Boolean,
  if $i = \max_{j \in G}\{j\}$
  then *true* else *false*
$Next(U, G)$, next task u, such that
  $rank(u, U) = rank(i, G)$

History variables:

$cv_i \in views \ (i \in \mathcal{P})$,
  initially $\forall i, \ cv_i = v_0$.
$\text{MSG}_i \in 2^{\mathcal{M}es} \ (i \in \mathcal{P})$,
  initially $\forall i, \ \text{MSG}_i = \emptyset$.

Transitions at i**:**

input $\text{REQUEST}_{q,i}$
Effect:
  $requests \leftarrow requests \cup \{q\}$

input $\text{NEWVIEW}(v)_i$
Effect:
  $G \leftarrow v.set$
  $A \leftarrow \emptyset$
  $Rnd \leftarrow 1$
  $Phase \leftarrow send$
  $cv := v$

output $\text{GP1SND}(m, j)_i$
Precondition:
  $Coordinator(j)$
  $Phase = send$
  $m = \langle i, D, R, Rnd\rangle$
Effect:
  $\text{MSG} := \text{MSG} \cup \{m\}$
  $Phase \leftarrow receive$

input $\text{GP1RCV}(\langle j, H, Q, round\rangle)_i$
Effect:
  $A \leftarrow A \cup \{\langle j, H, Q, round\rangle\}$
  $R \leftarrow R \cup Q$
  $D \leftarrow D \cup H$
  if $G = \{j : \langle j, *, *, Rnd\rangle \in A\}$
  then
    $Phase \leftarrow mcast$

output $\text{GPMSND}(m)_i$
Precondition:
  $Coordinator(i)$
  $m = \langle i, D, R, Rnd\rangle$
  $Phase = mcast$
Effect:
  $\text{MSG} := \text{MSG} \cup \{m\}$
  $Phase \leftarrow mrecv$

input $\text{GPMRCV}(\langle j, H, Q, round\rangle)_i$
Effect:
  $D \leftarrow D \cup H$
  $R \leftarrow R \cup Q$
  if $D = T$ then
    $Phase \leftarrow sleep$
  else
    if $rank(i, G) < |U|$ then
      $R \leftarrow R \cup \{Result(Next(U, G))\}$
      $D \leftarrow D \cup \{Next(U, G)\}$
    $Rnd \leftarrow Rnd + 1$
    $Phase \leftarrow send$

output $\text{REPORT}(results)_{q,i}$
Precondition:
  $T = D \wedge q \in requests$
  $results = R$
Effect:
  $requests \leftarrow requests - \{q\}$

FIG. 2. Algorithm *AX*.

cessor stops working as long as it knows of any undone tasks.

THEOREM 4.1
(Safety 1) For all states of any execution of Algorithm *AX* it holds that
$\forall i \in \mathcal{P} : D_i \neq T \Rightarrow Phase \neq sleep.$

PROOF. The proof follows by examination of the code of the algorithm, and more specifically from the code of the input action GPMRCV$(\langle j, H, Q, round\rangle)_i$. ∎

Note that the implication in Theorem 4.1 cannot be replaced by iff ($\Leftrightarrow$). This is because if $D_i = T$, we may still have $Phase \neq sleep$. This is the case where processor i becomes a member of a group in which the processors do not know all the results of all the tasks.

Next we show that if some processor does not know the result of some task, this is because it does not know that this task has been performed (Theorem 4.3 below). We show this using the history variables MSG$_i$ $(i \in \mathcal{P})$.

We define MSG$_i$ to be a history variable that keeps on track all the messages sent by processor $i \in \mathcal{P}$ in all GP1SND and GPMSND events of an execution of algorithm AX. Formally, in the effects of the GP1SND$(m, j)_i$ and GPMSND$(m)_i$ actions we include the assignment MSG$_i$:= MSG$_i \cup \{m\}$. Initially, MSG$_i = \emptyset$ for all i. We define $\mathcal{MSG}$ to be $\cup_{i\in\mathcal{P}}$ MSG$_i$.

LEMMA 4.2
If m is a message received by processor $i \in \mathcal{P}$ in a GP1RCV$(m)_i$ or GPMRCV$(m)_i$ event of an execution of algorithm *AX*, then $m \in \mathcal{MSG}$.

PROOF. Property 3 of the GCS (Section 2.2) requires that for every receive event there exists a preceding send event of the same message (the GCS does not generate messages). Hence, m must have been sent by some processor $q \in \mathcal{P}$ (possible $q = i$) in some earlier event of the execution. Messages can be sent only in GP1SND$(m, i)_q$ or GPMSND$(m)_q$ events. By definition, $m \in$ MSG$_q$. Hence, $m \in \mathcal{MSG}$. ∎

THEOREM 4.3
(Safety 2) For all states of any execution of Algorithm *AX*:
(a) $\forall t \in T,\ \forall i \in \mathcal{P} : result(t) \notin R_i \Rightarrow t \notin D_i$, and
(b) $\forall t \in T, \forall\langle i, D', R', Rnd\rangle \in \mathcal{MSG} : result(t) \notin R' \Rightarrow t \notin D'$.

PROOF. Let α be an execution of *AX* and α^k be the prefix of α up to the k^{th} state, i.e., $\alpha^k = s_0, e_1, s_1, e_2, \ldots, s_k$. The proof is done by induction on k.
Basis: $k = 0$. In s_0, $\forall i \in \mathcal{P}, D_i = \emptyset, R_i = \emptyset$ and $\mathcal{MSG} = \emptyset$.
Inductive hypothesis: For a state s_n such that $n \leq k$, $\forall t \in T$, $\forall i \in \mathcal{P} : result(t) \notin R_i \Rightarrow t \notin D_i$, and $\forall t \in T, \forall\langle i, D', R', Rnd\rangle \in \mathcal{MSG} : result(t) \notin R' \Rightarrow t \notin D'$.
Inductive step: $n = k + 1$. Consider the following seven types of actions leading to the state s_{k+1}:

1. $e_{k+1} =$ NEWVIEW$(v')_i$: The effect of this action does not affect the invariant. By the inductive hypothesis, in state s_{k+1}, the invariant holds.
2. $e_{k+1} =$ GP1SND$(m, j)_i$: Clearly, the effect of this action does not affect part (a) of the invariant but it affects part (b). Since $m = \langle i, D_i, R_i, Rnd\rangle$, by the

inductive hypothesis part (a), the assignment $m \in \mathcal{MSG}$ reestablishes part (b) of the invariant. Thus, in state s_{k+1}, the invariant is reestablished.

3. $e_{k+1} = \text{GP1RCV}(\langle j, H, Q, round\rangle)_i$: Processor i updates R_i and D_i according to Q and H respectively. The action is atomic, i.e., if R_i is updated, then D_i must be also updated. By Lemma 4.2, $\langle j, H, Q, round\rangle \in \mathcal{MSG}$. Thus, by the inductive hypothesis part (b), $\forall t \in T : result(t) \notin H \Rightarrow t \notin Q$. From the fact that D_i and R_i are updated according to H and Q respectively and by the inductive hypothesis part (a), in state s_{k+1}, the invariant is reestablished.
4. $e_{k+1} = \text{GPMSND}(m)_i$: Clearly, the effect of this action does not affect part (a) of the invariant but it affects part (b). Since $m = \langle i, D_i, R_i, Rnd\rangle$, by the inductive hypothesis part (a), the assignment $m \in \mathcal{MSG}$ reestablishes part (b) of the invariant. Thus, in state s_{k+1}, the invariant is reestablished.
5. $e_{k+1} = \text{GPMRCV}(\langle j, H, Q, round\rangle)_i$: By Lemma 4.2, $\langle j, H, Q, round\rangle \in \mathcal{MSG}$. By the inductive hypothesis part (b), $\forall t \in T : result(t) \notin H \Rightarrow t \notin Q$. Processor i updates R_i and D_i according to Q and H respectively. Since H and Q have the required property, by the inductive hypothesis part (a), the assignments to D_i and R_i reestablish the invariant.
   In the case where $D_i \neq T$, processor i performs a task according to the load balancing rule. Let $u \in T$ be this task. Because of the action atomicity, when processor i updates R_i with $result(u)$, it must also update D_i with u. Hence, in state s_{k+1}, the invariant is reestablished.
6. $e_{k+1} = \text{REQUEST}_{q,i}$: The effect of this action does not affect the invariant.
7. $e_{k+1} = \text{REPORT}(results)_{q,i}$: The effect of this action does not affect the invariant.

This completes the proof. ∎

5 Analysis of Algorithm *AX*

We express the work complexity of algorithm *AX* in the model $\mathcal{F}_{FM}$ as $W_{\mathcal{F}_{FM}}(N, P, r)$ $= W_{\mathcal{F}_{FM}}(N, P, f + m)$. The message complexity is expressed as $M_{\mathcal{F}_{FM}}(N, P, r) =$ $M_{\mathcal{F}_{FM}}(N, P, f + m)$. Our analysis focuses on assessing the impact of the fragmentation number f and the merge number m on the work and message complexity, and in the rest of this section for clarity we let $\mathcal{W}_{f,m}$ stand for $W_{\mathcal{F}_{FM}}(N, P, f + m)$, and $\mathcal{M}_{f,m}$ stand for $M_{\mathcal{F}_{FM}}(N, P, f + m)$.

5.1 Work Complexity

In this section we show the following result:

THEOREM 5.1
$\mathcal{W}_{f,m} \leq \min\{\, N \cdot f + N,\ N \cdot P \,\}$.

Observe that $\mathcal{W}_{f,m}$ does not depend on m (this of course does not imply that for any given execution, the work does not depend on merges). This observation substantiates the intuition that merges lead to a more efficient computation. We begin by providing definitions and proving several lemmas that lead to the above result.

DEFINITION 5.2
Let α^{μ} be any execution of algorithm *AX* in which all the processors learn the results of all tasks and that includes a merge of groups $g_1, \ldots, g_k$ into the group μ, where the processors in μ undergo no further view changes. We define $\bar{\alpha}^{\mu}$ to be the execution we derive by removing the merge from α^{μ} as follows:
(1) We remove all states and events that correspond to the merge of groups $g_1, \ldots, g_k$ into the group μ and all states and events for processors within μ.
(2) We add the appropriate states and events such that the processors in groups $g_1, \ldots,$ g_k undergo no further view changes and perform any remaining tasks.

DEFINITION 5.3
Let α^{φ} be any execution of algorithm *AX* in which all the processors learn the results of all tasks and that includes a fragmentation of the group φ to the groups $g_1, \ldots, g_k$ where the processors in these groups undergo no further view changes. We define $\bar{\alpha}^{\varphi}$ to be the execution we derive by removing the fragmentation from α^{φ} as follows:
(1) We remove all states and events that correspond to the fragmentation of the group φ to the groups $g_1, \ldots, g_k$ and all states and events of the processors within the groups $g_1, \ldots, g_k$.
(2) We add the appropriate states and events such that the processors in the group φ undergo no further view changes and perform any remaining tasks.

Note: In Definitions 5.2 and 5.3, we claim that we can remove states and events from an execution and add some other states and events to it. This is possible because if the processors in a single view installed that view and there are no further view changes, then the algorithm will continue making computation progress. So, if we remove all states and events corresponding to a view change, then the algorithm can always proceed as if this view change never occurred.

LEMMA 5.4
In algorithm *AX*, for any view v, including the initial view, if the group is not subject to any regroupings, then the work required to complete all tasks in the view is no more than $N - \max_{i \in v.set}\{|D_i|\}$, where D_i is the value of the state variable D of processor i at the start of its local round 1 in view v.

PROOF. In the first round, all the processors send messages to the coordinator containing D_i. The coordinator computes $\cup_{i \in v.set}\{D_i\}$ and broadcasts this result to the group members. Since the group is not subject to any regroupings, the number of tasks, t, that the processors need to perform is: $t = N - | \cup_{i \in v.set} \{D_i\}|$. In each round of the computation, by the load balancing rule, the members of the group perform distinct tasks and no task is performed more than once. Therefore, t is the work performed in this group. On the other hand, $\max_{i \in v.set}\{|D_i|\} \leq | \cup_{i \in v.set} \{D_i\}|$, thus, $t \leq N - \max_{i \in v.set}\{|D_i|\}$. ∎

In the following lemma, groups $\mu, g_1, \ldots, g_k$ are defined as in Definition 5.2.

LEMMA 5.5
Let α^{μ} be an execution of Algorithm *AX* as in Definition 5.2. Let W_1 be the work performed by the algorithm in the execution α^{μ}. Let W_2 be the work performed by Algorithm *AX* in the execution $\bar{\alpha}^{\mu}$. Then $W_1 \leq W_2$.

PROOF. For the execution α^μ, let W' be the work performed by the processors in $\mathcal{P} - \bigcup_{1\le i\le k}(g_i.set) - \mu.set$. Observe that the work performed by the processors in $\mathcal{P} - \bigcup_{1\le i\le k}(g_i.set)$ in the execution $\bar{\alpha}^\mu$ is equal to W'. The work that is performed by processor j in $g_i.set$ prior to the NEWVIEW$(\mu)_j$ event in α^μ, is the same in both executions. Call this work $W_{i,j}$. Define $W'' = \sum_{i=1}^{k}\sum_{j\in g_i.set} W_{i,j}$. Define $W = W' + W''$. Thus, W is the same in both executions, α^μ and $\bar{\alpha}^\mu$. Define W_μ to be the work performed by all processors in $\mu.set$ in execution α^μ.

For each processor j in $g_i.set$, let D_j be the value of the state variable D just prior to the NEWVIEW$(\mu)_j$ event in α^μ. For each g_i, define: $d_i = |\bigcup_{j\in g_i.set} D_j|$. Thus there are at least $N - d_i$ tasks that remain to be done in each g_i.

In execution $\bar{\alpha}^\mu$, the processors in each group g_i proceed and complete these remaining tasks. This requires work at least $N - d_i$. Define this work as W_{g_i}. Thus, $W_{g_i} \ge (N - d_i)$.

In execution α^μ, groups $g_1, \ldots, g_k$ merge into group μ. The number of tasks that need to be performed by the members of μ is at most $N - d_j$, where $d_j = \max_i\{d_i\}$ for some j. By Lemma 5.4, $W_\mu \le N - d_j$. Observe that:

$$W_1 = W + W_\mu \le W + N - d_j \le W + \sum_{i=1}^{k}(N - d_i) \le W + \sum_{i=1}^{k} W_{g_i} = W_2. \quad \blacksquare$$

In the following lemma, groups $\varphi, g_1, \ldots, g_k$ are defined as in Definition 5.3.

LEMMA 5.6
Let α^φ be an execution of Algorithm *AX* as in Definition 5.3. Let W_1 be the work performed by the algorithm in the execution α^φ. Let W_2 be the worked performed by Algorithm *AX* in the execution $\bar{\alpha}^\varphi$. Then $W_1 \le W_2 + W_3$, where W_3 is the work performed by all processors in $\bigcup_{1\le i\le k}(g_i.set)$ in the execution α^φ.

PROOF. Let W' be the work performed by all processors in $\mathcal{P} - \bigcup_{1\le i\le k}(g_i.set) - \varphi.set$ in the execution α^φ. Observe that the work performed by all processors in $\mathcal{P} - \varphi.set$ in the execution $\bar{\alpha}^\varphi$ is equal to W'. The work that is performed by processor j in $\varphi.set$ prior to the NEWVIEW$(g_i)_j$ event in α^φ, is the same in both executions. Call this work $W_{\varphi,j}$. Define $W'' = \sum_{j\in\varphi.set} W_{\varphi,j}$. Define $W = W' + W''$. Thus, W is the same in both executions, α^φ and $\bar{\alpha}^\varphi$. Define W_φ to be the work performed by all processors in $\varphi.set$ in execution $\bar{\alpha}^\varphi$. Let $W''' = W_\varphi - W''$. Observe that:

$$W_1 = W + W_3 \le W + W_3 + W''' = W_2 + W_3. \quad \blacksquare$$

LEMMA 5.7
$\mathcal{W}_{f,m} \le N \cdot P$.

PROOF. By the construction of algorithm *AX*, when processors are not able to exchange information about task execution due to regroupings, in the worst case, each processor has to perform all N tasks by itself. Since we can have at most P processors doing that, the work is: $\mathcal{W}_{f,m} \le N \cdot P$. ■

LEMMA 5.8
$\mathcal{W}_{f,m} \le N \cdot f + N$.

PROOF. By induction on the number of views, denoted by r, occurring in an execution. For a specific execution α_r with r views, let f_r be the fragmentation-number and m_r the merge-number.

Basis: $r = 0$. Since f_r and m_r must also be 0, the basis follows from Lemma 5.4.

Inductive hypothesis: Assume that for all $r \leq k$, $\mathcal{W}_{f_r,m_r} \leq N \cdot f_r + N$.

Inductive step: Need to show that for $r = k+1$, $\mathcal{W}_{f_{k+1},m_{k+1}} \leq N \cdot f_{k+1} + N$. Consider a specific execution α_{k+1} with $r = k+1$. Let $\Gamma_{\alpha_{k+1}}$ be the view-graph induced by this execution. The view-graph has at least one vertex such that all of its descendants are sinks (Fact 3.14). Let ν be such a vertex. We consider two cases.

Case 1: ν has a descendant μ that corresponds to a merge in the execution. Therefore all ancestors of μ in $\Gamma_{\alpha_{k+1}}$ have outdegree 1. Since μ is a sink vertex, the group that corresponds to μ performs all the remaining (if any) tasks and does not perform any additional work.

Let $\alpha_k = \bar{\alpha}^{\mu}_{k+1}$ (per Definition 5.2) be an execution in which this merge does not occur. In execution α_k, the number of views is k. Also, $f_{k+1} = f_k$ and $m_{k+1} = m_k + 1$. By inductive hypothesis, $\mathcal{W}_{f_k,m_k} \leq N \cdot f_k + N$. By Lemma 5.5, the work performed in execution α_{k+1}, is no worse than the work performed in execution α_k. The total work complexity is:

$$\mathcal{W}_{f_{k+1},m_{k+1}} \;\leq\; \mathcal{W}_{f_k,m_k} \;\leq\; N \cdot f_k + N \;=\; N \cdot f_{k+1} + N.$$

Case 2: ν has no descendants that correspond to a merge in the execution. Therefore, the group that corresponds to ν must fragment, say into q groups. These groups correspond to sink vertices in $\Gamma_{\alpha_{k+1}}$, thus they perform all the remaining (if any) tasks and do not perform any additional work.

Let $\alpha_{k+1-q} = \bar{\alpha}^{\nu}_{k+1}$ (per Definition 5.3) be an execution in which the fragmentation does not occur. In execution α_{k+1-q}, the number of views is $k+1-q \leq k$. Also, $f_{k+1-q} = f_{k+1} - q$ and $m_{k+1-q} = m_{k+1}$. By inductive hypothesis, $\mathcal{W}_{f_{k+1-q},m_{k+1-q}} \leq N \cdot f_{k+1-q} + N$. From Lemma 5.4, the work performed in each new group caused by the fragmentation is no more than N. Let W_σ be the total work performed in all q groups. Thus, $W_\sigma \leq qN$. By Lemma 5.6, the work performed in execution α_{k+1}, is no worse than the work performed in execution α_{k+1-q} and the work performed in all q groups. The total work complexity is:

$$\begin{array}{llll}
\mathcal{W}_{f_{k+1},m_{k+1}} & \leq \mathcal{W}_{f_{k+1-q},m_{k+1-q}} + W_\sigma & \leq N \cdot f_{k+1-q} + N + W_\sigma \\
& = N \cdot (f_{k+1} - q) + N + W_\sigma & \leq N \cdot (f_{k+1} - q) + N + qN \\
& = N f_{k+1} - qN + N + qN & = N \cdot f_{k+1} + N. & \blacksquare
\end{array}$$

The main result in Theorem 5.1 follows directly from Lemmas 5.7 and 5.8.

5.2 *Message Complexity*

In this section we show the following result:

THEOREM 5.9
$\mathcal{M}_{f,m} \;<\; 4(N \cdot f + N + P \cdot m)$

We start by showing several lemmas that lead to the message complexity result.

LEMMA 5.10
For algorithm *AX*, in any view v, including the initial view, if the group is not subject to any regroupings, and for each processor $i \in v.set$, D_i is the value of the state variable D at the start of its local round 1 in view v, then the number of messages M that are sent until all tasks are completed is $2(N-d) \leq M < 2(p+N-d)$ where $p = |v.set|$, and $d = |\bigcup_{i \in v.set} D_i|$.

PROOF. By the load balancing rule, the algorithm needs $\lceil \frac{N-d}{p} \rceil$ rounds to complete all tasks. In each round each processor sends one message to the coordinator and the coordinator responds with a single message to each processor. Thus, $M = 2p \cdot (\lceil \frac{N-d}{p} \rceil)$. Using the properties of the *ceiling*, we get: $2(N-d) \leq M < 2(p+N-d)$. ■

In the following lemma, groups $\mu, g_1, \ldots, g_k$ are defined as in Definition 5.2.

LEMMA 5.11
Let α^μ be an execution of Algorithm *AX* as in Definition 5.2. Let M_1 be the message cost of the algorithm in the execution α^μ. Let M_2 be the message cost of Algorithm *AX* in the execution $\bar{\alpha}^\mu$. Then $M_1 < M_2 + 2P$.

PROOF. For the execution α^μ, let M' be the number of messages sent by the processors in $\mathcal{P} - \bigcup_{1 \leq i \leq k}(g_i.set) - \mu.set$. Observe that the number of messages sent by the processors in $\mathcal{P} - \bigcup_{1 \leq i \leq k}(g_i.set)$ in the execution $\bar{\alpha}^\mu$ is equal to M'.

The number of messages sent by any processor j in $g_i.set$ prior to the NEWVIEW$(\mu)_j$ event in α^μ, is the same in both executions. Call this message cost $M_{i,j}$. Define $M'' = \sum_{i=1}^{k} \sum_{j \in g_i.set} M_{i,j}$. Define $M = M' + M''$. Thus, M is the same in both executions, α^μ and $\bar{\alpha}^\mu$. Define M_μ to be the number of messages sent by all processors in $\mu.set$ in execution α^μ.

For each processor j in $g_i.set$, let D_j be the value of the state variable D just prior to the NEWVIEW$(\mu)_j$ event in α^μ. For each g_i, define: $d_i = |\bigcup_{j \in g_i.set} D_j|$. Thus there are at least $N - d_i$ tasks that remain to be done in each g_i.

In execution $\bar{\alpha}^\mu$, the processors in each group g_i proceed and complete these remaining tasks. Let M_{g_i} be the number of messages sent by all processors in $g_i.set$ in order to complete the remaining tasks. By Lemma 5.10, $M_{g_i} \geq 2(N - d_i)$.

In execution α^μ, groups $g_1, \ldots, g_k$ merge into group μ. The number of tasks that need to be performed by the members of μ is at most $N - d_j$, where $d_j = \max_i\{d_i\}$ for some j. By Lemma 5.10, $M_\mu < 2(p + N - d_j)$, where $p = |\mu.set|$. Observe that:

$$\begin{array}{llll} M_1 & = M + M_\mu & < M + 2(p + N - d_j) \\ & \leq M + 2p + 2\sum_{i=1}^{k}(N - d_i) & \leq M + 2p + \sum_{i=1}^{k} M_{g_i} \\ & = M_2 + 2p & \leq M_2 + 2P. \end{array}$$
■

In the following lemma, groups $\varphi, g_1, \ldots, g_k$ are defined as in Definition 5.3.

LEMMA 5.12
Let α^φ be an execution of Algorithm *AX* as in Definition 5.3. Let M_1 be the message cost of the algorithm in the execution α^φ. Let M_2 be the message cost of Algorithm *AX* in the execution $\bar{\alpha}^\varphi$. Then $M_1 \leq M_2 + M_3$, where M_3 is the number of messages sent by all processors in $\bigcup_{1 \leq i \leq k}(g_i.set)$ in the execution α^φ.

PROOF. Fir the execution α^{φ}, let M' be the number of messages sent by the processors in $\mathcal{P} - \bigcup_{1\leq i\leq k}(g_i.set) - \varphi.set$. Observe that the number of messages sent by the processors in $\mathcal{P} - \varphi.set$ in the execution $\bar{\alpha}^{\varphi}$ is equal to M'.
The number of messages sent by processor j in $\varphi.set$ prior to the NEWVIEW$(g_i)_j$ event in α^{φ}, is the same in both executions. Call this message cost $M_{\varphi,j}$. Define $M'' = \sum_{j\in\varphi.set} M_{\varphi,j}$. Define $M = M' + M''$. Thus, M is the same in both executions, α^{φ} and $\bar{\alpha}^{\varphi}$. Define M_{φ} to be the number of messages sent by all processors in $\varphi.set$ in execution $\bar{\alpha}^{\varphi}$. Let $M''' = M_{\varphi} - M''$. Observe that:

$$M_1 = M + M_3 \leq M + M_3 + M''' = M_2 + M_3. \qquad \blacksquare$$

We now give the proof of Theorem 5.9. This is done by induction, similarly to the proof of Lemma 5.8.

PROOF. (For Theorem 5.9.) By induction on the number of views, denoted by r, occurring in any execution. For a specific execution α_r with r views, let f_r be the fragmentation number and m_r be the merge-number.

Basis: $r = 0$. Since f_r and m_r must also be 0, the basis follows from Lemma 5.10.

Inductive hypothesis: Assume that for all $r \leq k$, $\mathcal{M}_{f_r,m_r} < 4(N \cdot f_r + N + P \cdot m_r)$.

Inductive step: Need to show that for $r = k+1$, $\mathcal{M}_{f_{k+1},m_{k+1}} < 4(N \cdot f_{k+1} + N + P \cdot m_{k+1})$. Consider a specific execution α_{k+1} with $r = k+1$. Let $\Gamma_{\alpha_{k+1}}$ be the view-graph induced by this execution. The view-graph has at least one vertex such that all of its descendants are sinks (Fact 3.14). Let ν be such a vertex.
We consider two cases.

Case 1: ν has a descendant μ that corresponds to a merge in the execution. Therefore all ancestors of μ in $\Gamma_{\alpha_{k+1}}$ have outdegree 1. Since μ is a sink vertex, the group that corresponds to μ performs all the remaining (if any) tasks and no further messages are sent.

Let $\alpha_k = \bar{\alpha}^{\mu}_{k+1}$ (per Definition 5.2) be an execution in which this merge does not occur. In execution α_k, the number of new views is k. Also, $f_{k+1} = f_k$ and $m_{k+1} = m_k + 1$. By inductive hypothesis, $\mathcal{M}_{f_k,m_k} < 4(N \cdot f_k + N + P \cdot m_k)$. The total message complexity, using Lemma 5.11 is:

$$\begin{aligned}
\mathcal{M}_{f_{k+1},m_{k+1}} &< \mathcal{M}_{f_k,m_k} + 2P \\
&< 4(N \cdot f_k + N + P \cdot m_k) + 2P \\
&= 4(N \cdot f_{k+1} + N + P \cdot m_{k+1} - P) + 2P \\
&= 4Nf_{k+1} + 4N + 4Pm_{k+1} - 4P + 2P \\
&\leq 4(N \cdot f_{k+1} + N + P \cdot m_{k+1}).
\end{aligned}$$

Case 2: ν has no descendants that correspond to a merge in the execution. Therefore, the group that corresponds to ν must fragment, say into q groups. These groups correspond to sink vertices in $\Gamma\alpha_{k+1}$, thus they perform all of the remaining (if any) tasks and do not send any additional messages.

Let $\alpha_{k+1-q} = \bar{\alpha}^{\nu}_{k+1}$ (per Definition 5.3) be an execution in which the fragmentation does not occur. In the execution α_{k+1-q}, the number of new views is

$k+1-q \leq k$. Also, $f_{k+1-q} = f_{k+1} - q$ and $m_{k+1-q} = m_{k+1}$. By inductive hypothesis, $\mathcal{M}_{f_{k+1-q}, m_{k+1-q}} < 4(N \cdot f_{k+1-q} + N + P \cdot m_{k+1-q})$. From Lemma 5.10, the message cost in each new group caused by a fragmentation is no more than $4N$. Let M_σ be the total number of messages sent in all q groups. Thus, $M_\sigma \leq 4qN$. By Lemma 5.12, the number of messages sent in execution α_{k+1}, is less than the number of messages sent in execution α_{k+1-q} and the number of messages sent in all q groups. The total message complexity is:

$$\begin{aligned}
\mathcal{M}_{f_{k+1}, m_{k+1}} &\leq \mathcal{M}_{f_{k+1-q}, m_{k+1-q}} + M_\sigma \\
&< 4(N \cdot f_{k+1-q} + N + P \cdot m_{k+1-q}) + M_\sigma \\
&= 4(N \cdot f_{k+1} - qN + N + P \cdot m_{k+1}) + M_\sigma \\
&\leq 4N f_{k+1} - 4qN + 4N + 4P m_{k+1} + 4qN \\
&= 4(N \cdot f_{k+1} + N + P \cdot m_{k+1}).
\end{aligned}$$

■

5.3 *Analysis for the Fragmentation Adversary*

We express the work complexity of algorithm *AX* in the model $\mathcal{F}_F$ as $W_{\mathcal{F}_F}(N, P, r) = \mathcal{W}_f$ and the message complexity as $M_{\mathcal{F}_F}(N, P, r) = \mathcal{M}_f$ (note that $r = f$ for $\mathcal{F}_F$). The following corollary is derived from Theorems 5.1 and 5.9.

COROLLARY 5.13
$\mathcal{W}_f \leq min\{N \cdot f + N,\ N \cdot P\}$ and $\mathcal{M}_f < 4(N \cdot f + N)$.

In the failure model of [8] a group is not allowed to "fragment" into a single group with the same membership. Such fragmentation is allowed by our definition of $\mathcal{F}_F$. In order to compare our results with the results of [8], we define a more restricted adversary $\mathcal{F}'_F$ that requires that any group may only fragment into 2 or more other groups. Clearly $\mathcal{F}'_F \subseteq \mathcal{F}_F$, and from Corollary 5.13 we have the following.

COROLLARY 5.14
$W_{\mathcal{F}'_F}(N, P, f) = O(N \cdot f + N)$ and $M_{\mathcal{F}'_F}(N, P, f) = O(N \cdot f + N)$.

In the rest of this section we deal with the model $\mathcal{F}'_F$. Our definition of the fragmentation-number f is slightly different from the definition of fragmentation failures f' in [8]. When a group fragments into k groups, f is defined to be equal to k, but f' is defined to be equal to $k-1$. The next Lemma relates f and f'.

LEMMA 5.15
If f is the fragmentation-number and f' the number of fragmentation failures as defined in [8], then $f' < f < 2f'$.

PROOF. Assume that k fragmentations occur. Enumerate the fragmentations arbitrarily. Let the number of the new views in the i^{th} fragmentation be f_i. By the definition of f'_i, $f'_i = f_i - 1$. Thus, $f'_i + 1 = f_i$ which implies that $f_i < f'_i + f'_i = 2f'_i$. But $f' = \sum_{i=1}^{k} f'_i$ and $f = \sum_{i=1}^{k} f_i$. Hence, $f < 2f'$. Now observe that, $f' = \sum_{i=1}^{k} f'_i = \sum_{i=1}^{k} (f_i - 1) = \sum_{i=1}^{k} f_i - k = f - k$. Therefore $f > f'$. ■

In [8] the work is counted in terms of the rounds executed by the processors. In our analysis we count only the number of task executions (including redundancies).

However in our algorithm, for as long as any tasks remain undone in a given group, the processors perform the tasks in rounds, except for the last round. Therefore the difference in work complexity for these two algorithms is at most $f \cdot N$. Thus the different definitions of f, f' and work are subsumed in the big-oh analysis, and without substantial variation in the constants. On the other hand, the message complexity of our algorithm, as shown in Corollary 5.14, is substantially better than the at least quadratic message complexity of the algorithm from [8].

6 Conclusion

We have considered the problem of performing a set of N tasks on a set of P cooperating message-passing processors, where the processors must perform all tasks and learn the results of the tasks, subject to dynamically changing group memberships. To analyze our algorithm we introduced view-graphs – digraphs that we use to represent and analyze changes of processors' views in executions. We believe that our view-graph approach is general and that it can be used to study other dynamic group reconfiguration patterns and related problems.

Acknowledgements: We thank Idit Keidar for several helpful discussions and the anonymous referees for their comments that helped us improve the quality of the presentation.

This work was in part supported by the NSF Grant 9988304, the NSF ITR Grant 0121277 and a grant from AFOSR; the second author was also supported by a NSF CAREER Award 9984778.

References

[1] O. Babaoglu, R. Davoli, L. Giachini and M. Baker, "Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems", in *Proc. of Hawaii International Conference on Computer and System Science*, volume II, pp 612–621, 1995.

[2] K.P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, Los Alamitos, CA, 1994.

[3] B. Chlebus, R. De Prisco and A. Shvartsman, "Performing tasks on restartable message-passing processors", in *Distributed Computing*, vol. 14, pp. 49-64, 2001.

[4] *Comm. of the ACM*, Special Issue on Group Communication Services, vol. 39, no. 4, 1996.

[5] R. De Prisco, A. Fekete, N. Lynch and A. Shvartsman, "A Dynamic View-Oriented Group Communication Service", in *Proc. of 16th ACM Symp. on Principles of Distributed Computing*, 1998.

[6] R. De Prisco, A. Mayer, and M. Yung, "Time-Optimal Message-Efficient Work Performance in the Presence of Faults", in *Proc. 13th ACM Symp. on Principles of Distributed Comp.*, pp. 161–172, 1994.

[7] D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communications", *Comm. of the ACM*, vol. 39, no. 4, pp. 64–70, 1996.

[8] S. Dolev, R. Segala and A. Shvartsman, "Dynamic Load Balancing with Group Communication", in *Proc. of the 6th International Colloquium on Structural Information and Communication Complexity*, 1999.

[9] C. Dwork, J. Halpern, O. Waarts, "Performing Work Efficiently in the Presence of Faults", *SIAM Journal on Computing*, vol. 27, no. 5, pp. 1457–1491, 1998.

[10] P. Ezhilchelvan, R. Macedo and S. Shrivastava "Newtop: A Fault-Tolerant Group Communication Protocol", in *Proc. of IEEE Int-l Conference on Distributed Computing Systems*, pp 296–306, 1995.

[11] A. Fekete, N. Lynch, and A.A. Shvartsman, "Specifying and Using a Group Communication Service", *ACM Transactions on Computer Systems*, vol. 19, pp. 171–216, 2001.

[12] Z. Galil, A. Mayer, and M. Yung, "Resolving Message Complexity of Byzantine Agreement and Beyond", in *Proc. 36th IEEE Symposium on Foundations of Computer Science*, pp. 724–733, 1995.

[13] C. Georgiou and A. Shvartsman,"Cooperative Computing with Fragmentable and Mergeable Groups", in *Proc. of 7th International Colloquium on Structural Information and Communication Complexity*, pp. 141–156, 2000.

[14] M. Hayden, Doctoral Thesis, *The Ensemble System*, TR98-1662, Cornell University, 1998.

[15] P. Kanellakis and A. Shvartsman, *Fault-Tolerant Parallel Computation*, Kluwer Academic Publishers, ISBN 0-7923-9922-6, 1997.

[16] R. Khazan, A. Fekete and N. Lynch, "Group Communication as a base for a Load-Balancing, Replicated Data Service", in *Proc. of the 12th International Symposium on Distributed Computing*, 1998.

[17] E. Y. Lotem, I. Keidar, and Danny Dolev, "Dynamic Voting for Consistent Primary Components", *Proc. of the 16th Annual ACM Symp. on Principles of Distributed Computing*, pp. 63–71, 1997.

[18] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[19] N.A. Lynch and M.R. Tuttle, "An Introduction to Input/Output Automata", *CWI Quarterly*, vol.2, no. 3, pp. 219–246, 1989.

[20] G. Malewicz, A. Russell and A.A. Shvartsman, "Distributed Cooperation During the Absence of Communication", in *Proc. 14th International Conference on Distributed Computing*, LNCS Vol. 1914, pp. 119–133, 2000. (Preliminary version: Brief announcement. *19th ACM Symposium on Principles of Distributed Computing*, 2000.)

[21] L.E. Moser, P.M. Melliar-Smith, D.A. Agarawal, R.K. Budhia and C.A. Lingley-Papadopolous, "Totem: A Fault-Tolerant Multicast Group Communication System", *Comm. of the ACM*, vol. 39, no. 4, pp. 54–63, 1996.

[22] S. Mishra, L.L. Peterson and R.D. Schlichting, "Consul: A Communication Substrate for Fault-Tolerant Distributed Programs", TR 91-32, dept. of Computer Science, University of Arizona, 1991.

[23] R. van Renesse, K.P. Birman and S. Maffeis, "Horus: A Flexible Group Communication System", *Comm. of the ACM*, vol. 39, no. 4, pp. 76–83, 1996.

[24] M. Saks, N. Shavit and H. Woll, "Optimal time randomized consensus – making resilient algorithms fast in practice", in *Proc. of the 2nd ACM-SIAM Symp. on Discrete Algorithms*, pp. 351–362, 1991.

[25] J. Sussman and K. Marzullo, "The Bancomat Problem: An Example of Resource Allocation in a Partitionable Asynchronous System", in *Proc of 12th Int-l Symp. on Distributed Computing*, 1998.

[26] R. Vitenberg, I. Keidar, G. V. Chockler and D. Dolev, "Group Communication Specifications: A Comprehensive Study", Technical Report CS99-31, Institute of Computer Science, The Hebrew University of Jerusalem, September 1999. (Also Technical Report MIT-LCS-TR-790, Laboratory for Computer Science, M.I.T., and Technical Report CS0964, Computer Science Department, the Technion, Haifa, Israel.)