# Dynamic Load Balancing with Group Communication*

Shlomi Dolev* Roberto Segala‡ Alex Shvartsman§

October 19, 1999

**Abstract**

This work considers the problem of efficiently performing a set of tasks using a network of processors in the setting where the network is subject to dynamic reconfigurations, including partitions and merges. A key challenge for this setting is the implementation of dynamic load balancing that reduces the number of tasks that are performed redundantly because of the reconfigurations. We explore new approaches for load balancing in dynamic networks that can be employed by applications using a group communication service. The group communication services that we consider include a membership service (establishing new groups to reflect dynamic changes) but does not include maintenance of a primary component. For the $n$-processor, $n$-task load balancing problem defined in this work, the following specific results are obtained.

For the case of fully dynamic changes including fragmentation and merges we show that the termination time of any on-line task assignment algorithm is greater than the termination time of an off-line task assignment algorithm by a factor greater than $n/12$.

We present a load balancing algorithm that guarantees completion of all tasks in *all* fragments caused by partitions with work $O(n + f \cdot n)$ in the presence of $f$ fragmentation failures.

We develop an effective scheduling strategy for minimizing the task execution redundancy and we prove that our strategy provides each of the $n$ processors with a schedule of $\Theta(n^{1/3})$ tasks such that at most *one* task is performed redundantly by *any* two processors.

Keywords: Load balancing, scheduling, dynamic networks, group communications.

## 1 Introduction

The problem of 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. This problem has been studied in a variety of setting, e.g., in shared-memory models [15] and message-passing models [10, 8]. In this work we consider this problem in the partitionable distributed setting where the computation can take advantage of group communication services and where the processors

[1]Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. Email: dolev@cs.bgu.ac.il. Part of this research was done while visiting the Laboratory of Computer Science at MIT.

[3]Dip. di Scienze dell'Informazione, Università di Bologna, Italy. Email: segala@cs.unibo.it.

[4]Dept. of Computer Science and Engineering, 191 Auditorium Rd., U-155, University of Connecticut, Storrs, CT 06269, USA and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. Email: alex@cse.uconn.edu. Part of this work was supported by a grant from AFOSR and the GTE Laboratories.

*The preliminary version of this works appears in the proceedings of the *6th International Colloquium on Structural Information and Communication Complexity (SIROCCO'99)*, 1999.

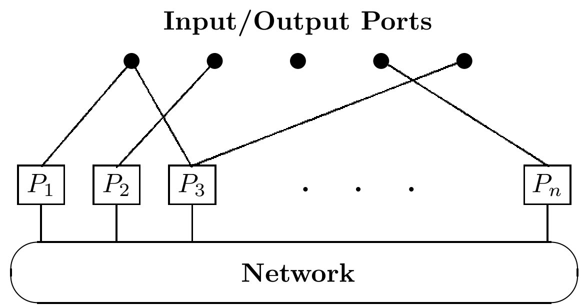


Figure 1: The distributed system and its input/output setting

have to perform the tasks efficiently even if they have to resort to scheduling the tasks in isolation due to network partition.

Group communication services can be used as effective building blocks for constructing fault-tolerant distributed applications [1]. The services enable the application components at different processors to operate collectively as a group, using the service to multicast messages. For applications involving coherent data, it is important to know when a processor has a view of the current group membership that is primary. Maintaining a primary group is one of the most sophisticated tasks of the group communication service. In a dynamic network environment the primary group will at times include only a portion of the distributed system. Thus in the cases where the computation has to be carried out in the primary group, only a fraction of the computation power of the distributed system is effectively used. However, there are settings in which *any group* of processors may meaningfully carry on with the computation irrespective of any other groups. For example, this is the case when a set of tasks, whose effects are idempotent, i.e., executing a task more than once yields identical results, needs to be performed in a distributed system. A simple example of this occurs when a collection of print servers is charged with the task of printing a set of reports. In a more dramatic setting suggested in [10], the tasks may consist of shutting a set of valves on a nuclear reactor.

In this work we investigate a new approach whose goal is to utilize the resources of *every component* of the system during the entire computation. We consider the problem in the following setting: a set of tasks must be performed by a distributed system (the tasks to be performed by the system may be submitted via the input ports. To simplify our presentation, we assume that the set of tasks has already been submitted). Group communication is used to coordinate the execution of the tasks. The requests for the tasks' results are submitted via the input/output ports. Once the results are known, the processors respond (see Figure 1). The main point in prescribing such input/output setting is that requests for results may be submitted externally to *any* processor. Our focus is on the investigation of effective load balancing schemes that lead to efficient execution of the set of tasks in such settings. Thus, we suggest a best effort approach, namely, an approach in which every processor that receives a request for results will eventually be able to respond with the complete set of results.

**Our contributions.** We study the problem of performing a set of tasks reliably and in parallel using multiple processors in the setting of message-passing processors that are interconnected by a network, which is subject to partitions and merges. We seek distributed solutions to the problem and we assume that computation is more expensive than communication. This assumption forces us to seek solutions that are more efficient than the trivial solutions, in which each processor performs each task. To assess the efficiency of solutions we use the complexity measure of work that accounts for each task performed by the processors including the tasks that are performed redundantly.

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 a client of the required computation to query any processor for 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. In other words, in this setting it is no longer sufficient to know that each of the tasks have been performed. It is also necessary for each processor to learn the results. In this paper we present the following results.

We show in Section 3 that developing efficient solutions for our model is difficult. For the problem of performing $n$ tasks on $n$ processors we present a *linear* (in the number of processors) lower bound for the worst case competitive ratio of the termination time of any on-line algorithm relative to an off-line algorithm. This competitive ratio is for the adversary that may cause arbitrary partitions and merges of the original network. We make no specific assumptions about the group communication service used.

The linear lower bound result suggests that to achieve more efficient load balancing, we need to limit the power of the adversary. In Section 4 we consider a setting with a restricted adversary that can dynamically cause *fragmentation failures*, i.e., the adversary can partition any existing connected component into two or more smaller components. We present and analyze an algorithm that relies on a group communication service. For this setting our load balancing algorithm for $n$ processors guarantees completion in all network fragments, and with the total work $O(n + f \cdot n)$ in the presence of any $f$ fragmentation failures. Note that this result also holds if we consider processor stop-failures since stopped processors can be modeled by isolating such processors from all other groups of processors.

The linear lower bound for the competitive ratio also shows that an on-line algorithm cannot do much better than the trivial solution in which every processor behaves as if it is a singleton and executes the entire set of tasks. With this in mind, in Section 5 we present an effective scheduling strategy for minimizing the task execution redundancy. We prove that for $n$ processors and $n$ tasks it is possible to schedule $\Theta(n^{1/3})$ tasks for each processor with at most *one* overlap in task executions. This means that using our algorithm, any two isolated processors can each perform up to $n^{1/3}$ tasks such that if the processors are merged into a group after $n^{1/3}$ such steps, then there is at most one task that is performed redundantly by the two processors.

**Related work.** Group communication services 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 group communication services are found in Isis [5], Transis [9], Totem [18], Newtop [11], Relacs [2], Horus [20] and Ensemble [4]. Examples of recent work dealing with primary groups are [7, 16]. An example of an application using a group communication service for load balancing is by Fekete, Khazan and Lynch [12]. To evaluate the effectiveness of partitionable group communication services, Sussman and Marzullo [23] proposed a measure (*cushion*) precipitated by a simple partition-aware application. Babaoglu et al. [3] study systematic support for partition awareness based on group communication services in a wide range of application areas, including applications that require load balancing. The main focus of the paper is the simplicity of implementing any load balancing policy within the group membership paradigm rather than the study of balancing policies that lead to good performance.

Our definition of work follows that of Dwork, Halpern and Waarts [10]. Our fragmentation model of failures 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 [6, 8, 10, 14], 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 [21] (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.

In Section 2 we present the problem and define our model, measures of efficiency and the group communication service. A lower bound on the competitive ratio is presented in Section 3. An algorithm for the fragmentation model is presented and analyzed in Section 4. Task scheduling algorithms for minimizing redundant work are in Section 5.

# 2 Problem Statement and Definitions

A distributed system consists of $n$ processors $(P_1, P_2, \ldots, P_n)$ connected by communication links. Each processor $P_i$ has a unique identifier. In Section 3 and Section 5 we assume that the identifiers are in the set $\{1, 2, \ldots, n\}$. At any given time a communication link may be operational or faulty. Faulty communication links can partition the system into several connected components. The recovery of the links may merge separated connected components into a single component. Link failures and recoveries trigger group membership activity to establish eventually a group for every connected component. The group membership service is used by the processors in the group to coordinate load balancing of task execution.

A set of tasks $T$ is to be executed by the distributed system. Processors receive $T$ from input ports and communicate $T$ to their group members. (Thus at the start of the computation, $T$ is known to all processors.) For the sake of simplicity of presentation we assume that the number of tasks in $T$ is exactly $n$, the number of processors in the system. Our results naturally extend to any $c \cdot n$ number of tasks $(c > 1)$ by either creating task groups of $c$ tasks in each group, or considering $c$ instances of the problem.

## 2.1 Performance Measures

The algorithms that we present in this paper are asynchronous. However, in order to study the performance of an asynchronous algorithm, we measure properties that are independent of time, and we study time bounds under some additional assumptions on the timings of the messages that are sent. In this paper we define a round based measure of the total work performed by the processors, and we study performance under the assumption that messages are delivered within time 1.

We define completion and termination times of a computation.

**Definition 2.1** *Given a set of processors and a set of tasks, the* completion time *of a computation is the minimal time at which every task is executed at least once.*

**Definition 2.2** *Given a set of processors and a set of tasks, the* termination time *of a computation is the time it takes for every processor to know the task execution results of all the tasks.*

From the above definitions it easy to see that *completion time* bounds, from below, the *termination time* for any computation. Our performance measures are based on a measure of the number of failures that occur within a computation. For the algorithm in Section 4 we consider only the *fragmentation failures*. In this setting, the initial group of $n$ processors is dynamically partitioned by failures into several fragments. The system begins with the initial *fragment* containing all $n$ processors, and each fragmentation failure "splits off" a fragment from an existing fragment.

**Definition 2.3** *For a computation in the fragmentation model that begins with $c_1$ fragments and terminates with $c_2$ fragments define the* number of failures $f$ *to be* $c_2 - c_1$.

Since fragments never merge, the number of fragmentation failures $f$ is at most $n-1$. Members of distinct fragments, existing concurrently, cannot communicate, and our model allows for processors in different fragments to terminate independently. Processors spend their lives communicating and working. We structure the lives of processors in terms of *rounds*. During a round, a processor may send a multicast message, receive some messages and perform a task. Within a specific fragment, each processor is charged for each round of the computation.

**Definition 2.4** *For a computation that terminates, we define* work *to be* $\sum_{1 \leq i \leq n} R_i$, *where* $R_i$ *is the number of rounds performed by processor* $i$.

In this work we do not explicitly deal with processor failures. However the definitions apply to, and the complexity results hold for the model that includes processor stop-failures. A processor that stops is modeled as a processor that is isolated from all others in a singleton group. Since a stopped processor does no work, it cannot increase the work complexity.

### 2.2 A Group Communication Service

We assume a virtual synchronous (or view synchronous) group communication service. The service is used to communicate information concerning the executed tasks once a new group is established. Each connected component of the system is an independent group that executes the (remaining) tasks in $T$ until the group is ready to output the final result. During the execution, the group communication service is used by the processors to notify each other of the results of task executions. Upon completion of the entire set of tasks the processors in the group supply the results to any external clients via the input/output ports.

The virtual synchronous service (or view synchronous service) that we rely on provides the following basic operations:

GPSND(*message*) The GPSND primitive lets a processor multicast a message to the members of the current group. The messages are guaranteed to be delivered unless a group change occurs. Messages are delivered in the same group they are sent in.

GPRCV(*message*) The GPRCV primitive enables a processor to receive multicasts from other processors in the current group view. (We do not require that message deliveries are ordered within a view.)

NEWVIEW($\langle id, set \rangle$) The NEWVIEW primitive tells a processor that a dynamic change caused a new group to be formed and it informs the processor of the identifier of the new group and the set of the identifiers of the processors in the group.

Figure 4 in the Appendix gives a formal specification (using I/O automata [17, 19]) of a view synchronous service that supports the operations we need. This service is the VS (view-syncrhonous) service from [13]. This service provides a total order on the messages sent in each view, and each processor receives some prefix of this order in the view. The group communication service sufficient for our needs is provided by several other existing specifications (cf. [2, 9, 20]). In the context of this work, our focus is not on the features provided by group communication services, but on the complexity of computation involving load balancing in the presence of commonly occuring group reconfigurations, e.g., group fragmentations and merges. In algorithm specification in Section 4 we assume that the group communication service is specified using Input/Output Automata, e.g., as in [7, 13].

## 3 Competitive Ratio for Dynamic Networks

In a fully dynamic network the system is subject to splits and merges, and the performance of the system may be drastically influenced by the exact pattern of such dynamic changes. A classical approach for evaluating an algorithm under such uncertain conditions is the competitive analysis proposed by Sleator and Tarjan in [22].

In this section we study the competitive ratio for the $n$-task assignment problem. The choice of the dynamic changes is a major parameter in computing a lower bound for the competitive ratio. For example under the assumption that the system is connected during the entire execution, there exists an optimal on-line (and an off-line) algorithm with completion and termination time 1. In this algorithm each processor, $P_i$, executes the $i$'th task first and reports the result to the other processors. In the other extreme when the system consists of $n$ singletons, there exists an optimal on-line (and off-line) algorithm with completion time 1 and termination time $n$. In this algorithm each processor, $P_i$, first executes the $i$'th task (thus the completion time is 1) and then the rest of the tasks (say by the order of their indices). The optimality of the above algorithms is due to the fact that any off-line algorithm does not performs better under the same partition pattern.

Next we present a lower bound for the worst case ratio of the termination time of an on-line task assignment algorithm versus the termination time of an off-line task assignment algorithm. Before we present the lower bound let us remark that it is easy to achieve completion time 1 when the number of the processors that participate is equal to the number of tasks. Completion time 1 is achieved by every algorithm in which each processor, $P_i$, executes the $i$th task first.

**Theorem 3.1** *There exists a group split and merge pattern for which the termination time of any on-line task assignment algorithm is greater than the termination time of an off-line task assignment algorithm by a factor greater than $n/12$.*

**Proof.** In the beginning the system is fully connected and the information concerning the tasks to be executed is sent to every processor. Then the processor set is partitioned into four groups $G_1, G_2, G_3, G_4$, with equal number of processors in each group. This partition is done before any of the results of the task execution is exchanged by the processors. "Notify" the on-line algorithm that two pairs of groups will be merged ($G_x$ with $G_y$ and $G_u$ with $G_v$) after the first two steps, without identifying their indices[1]. Also notify the on-line algorithm that immediately after the groups are combined and the information concerning the task executed is exchanged, the groups will be separated to singletons.

Note first that the off-line algorithm can terminate by the end of the group merge, since the identities of the groups to be merged are known to the off-line algorithm. Specifically and without

[1]Note that in reality the processors that execute the on-line algorithm do not know the partition pattern of the system nor the future dynamic changes, however we are interested in lower bounds for the competitive ratio, thus if we give a partial knowledge to the processors and still conclude a long termination time then we achieve our goal.

loss of generality, if the off-line algorithm knows that $G_1$ will be merged with $G_2$, the processors in $G_1$ may execute the tasks $1 \cdots n/4$ and then $n/2+1 \cdots 3n/4$ while the processors in $G_2$ execute the tasks $n/4+1 \cdots n/2$ and then $3n/4+1 \cdots n$. Once $G_1$ and $G_2$ merge and exchange the execution results, the processors in $G_1$ and $G_2$ may terminate. A similar argument holds for $G_3$ and $G_4$.

On the other hand, assume that the on-line algorithm assigns to each of $G_2, G_3$ and $G_4$ the tasks that are not executed by $G_1$. Then the merged group (say $G_1$ and $G_3$) that contains $G_1$ may be done with the tasks, however the other merged group (say $G_2$ and $G_4$) will have to execute half of the work, namely, the tasks executed by $G_1$ in the previous two steps. Since the processors (of the merged group formed by $G_2$ and $G_4$, in our example) are immediately partitioned into singleton groups, each will have to execute $n/2$ tasks, which leads to at least a linear in $n$ termination time.

Assume that the smallest set of tasks executed by both $G_1$ and another group $G_\ell$, $2 \leq \ell \leq 4$, does not include exactly one task, then the merged pair of groups that does not include $G_1$, has at least $n/2-2$ tasks to execute. Similarly, when the smallest set of tasks executed by $G_1$ with another group $G_\ell$, $2 \leq \ell \leq 4$, does not include $m$ tasks, then the merged pair of groups that does not include $G_1$, has at least $n/2-2m$ tasks to execute. The pair of groups that includes $G_1$ must execute at least $m$ tasks — since at least $m$ tasks are executed twice. Thus, the minimal possible number of tasks left to execute by either the group in which the processors in $G_1$, or the the processors of the other group participate is obtained from the equation $m = n/2 - 2m$, thus $m = n/6$. We can conclude that the competitive ratio is at least $(n/6+2)/2 = n/12+1 = \Omega(n)$. □

The linear ratio in the above result shows that an on-line algorithm cannot do much better than a trivial solution in which every processor behaves as if it is a singleton group and executes the entire set of tasks. With this in mind, we present in the next two sections first a scheduling algorithm for network fragmentation failures, and then a scheduling algorithm that minimizes redundant task executions even if processors may have to work initially in isolation from one another and then subsequently be merged into larger groups.

# 4 Load Balancing and Fragmentations

We now consider the setting with *fragmentation* failures and present a strategy for efficient task scheduling. We present this in terms of the algorithm that relies on a group communication service. We call it *algorithm AF*. The basic idea of the algorithm is that each processor performs (remaining) tasks according to a permutation until it learns that all tasks have been performed. The permutations are established by a global load balancing rule when there are no fragmentation failures. A processor performs tasks according to an arbitrary local rule when fragmentations do occur. We state the algorithm as a protocol that uses the group communication service described in Section 2.2. For completeness, we provide the code of a service sufficient for our needs, the VS service [13], in the Appendix.

**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 $k$ be the rank of $i$ in $G$ sorted in ascending order. Our *load balancing rule* is that processor $i$ performs the task $k \bmod |U|$, where $U$ is the number of remaining tasks.

**Algorithm structure.** The algorithm code is given in Figure 2 using I/O automata notation [19].

**Data types:**

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

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

**States:**

$T \in 2^{\mathcal{T}}$, set of $n = |T|$ tasks
$D \in 2^{\mathcal{T}}$, set of done tasks, initially $\emptyset$
$R \in 2^{\mathcal{R}}$, set of results, initially $\emptyset$
$G \in 2^{\mathcal{P}}$, group members, initially $\mathcal{P}$
$M \in 2^{\mathcal{M}}$, messages, initially $\emptyset$
$\mathit{Rnd} \in \mathbf{N}$, round number, initially 0
$\mathit{Phase} \in \{\mathit{send}, \mathit{receive}, \mathit{stop}\}$, initially *send*
$\mathit{requests} \in 2^{\mathcal{IO}}$, set of ports, initially $\emptyset$

**Derived variables:**
$U : T - \cup\{E : \langle *, *, E, \mathit{Rnd}\rangle \in M\}$, reported remaining tasks
$t$ : let $k$ be the rank of $i$ in $G$ sorted in ascending order, then $t$ is the id of the task whose rank is ($k$ mod $|U|$) in $U$ sorted by id

**Transitions at $i$:**

**input** $\mathrm{REQUEST}_{q,i}$
Effect:
  $\mathit{requests} \leftarrow \mathit{requests} \cup \{q\}$

**input** $\mathrm{NEWVIEW}(v)_i$
Effect:
  $G \leftarrow v.\mathit{set}$
  if $D \neq T$ then
    $s \leftarrow$ if $t \in D$
      then some task in $T - D$
      else $t$
    $R \leftarrow R \cup \{\mathit{Result}(s)\}$
    $D \leftarrow D \cup \{s\}$
  $M \leftarrow \emptyset$
  $\mathit{Rnd} \leftarrow 0$
  $\mathit{Phase} \leftarrow \mathit{send}$

**output** $\mathrm{GPSND}(m)_i$
Precondition:
  $\mathit{Phase} = \mathit{send}$
  $m = \langle i, D, R, \mathit{Rnd}+1\rangle$
Effect:
  $\mathit{Rnd} \leftarrow \mathit{Rnd}+1$
  $\mathit{Phase} \leftarrow \mathit{receive}$

**input** $\mathrm{GPRCV}(\langle j, Q, E, \mathit{round}\rangle)_i$
Effect:
  $M \leftarrow M \cup \{\langle j, Q, E, \mathit{round}\rangle\}$
  $R \leftarrow R \cup Q$
  $D \leftarrow D \cup E$
  if $G = \{j : \exists_{Q',E'} : \langle j, Q', E', \mathit{Rnd}\rangle \in M\}$ then
    if $D \neq T$ then
      $R \leftarrow R \cup \{\mathit{Result}(t)\}$
      $D \leftarrow D \cup \{t\}$
    if $T = \cap\{E : \langle *, *, E, \mathit{Rnd}\rangle \in M\}$ then
      $\mathit{Phase} \leftarrow \mathit{stop}$
    else
      $\mathit{Phase} \leftarrow \mathit{send}$

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

Figure 2: Algorithm *AF*.

The algorithm uses the group communication service to structure its computation in terms of *rounds* numbered sequentially within each group view.

Rounds numbered 0 correspond to group reconfigurations. If a fragmentation occurs, the processor receives the new set of members from the group membership service. The processor performs one task among those it believes are not done, and starts the next round. 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. In each round ($Rnd > 0$), each processor reports $D$ and $R$ to the members of $G$, collects such reports from other processors, updates $D$ and $R$, and performs one task according to the load balancing rule.

For generality, we assume that multicast messages may be delivered out of order with respect to the rounds. The set of messages within the current view is saved in the local variable $M$. 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.

When requests for computation results arrive from a port $q$, each processor keeps track of this in a local variable *requests*, and, when all results are known, sends the results to the port.

**Analysis of algorithm** $AF$**.** We now determine the worst-case work of the algorithm as a function of the initial number of processors $n$ (we are assuming that initially there is a single task per processor), and of the number of fragmentation failures $f$. We assume that a failure causes no more than one new view to be installed at each member of the group that fragments. We start by showing algorithm termination.

**Lemma 4.1** *In algorithm AF, each processor terminates having performed $O(n)$ tasks and executing $O(n)$ rounds.*

**Proof.** Since each processor keeps track of tasks performed in variable $D$, it never performs more than $n$ tasks. In each round without view changes, each processor performs at least one task that it does not have the results for. Since no processor performs the same task twice, there are at most $n$ such rounds.

Additional rounds may be executed due to fragmentations. Even if a particular processor has all the results, it must ensure that all members of the new group know the results[2]. Since there are at most $n-1$ fragmentations, the number of iterations attributable to view changes is also bounded by $n$. □

We define *complete* rounds for a view $v$ to be the rounds during which all processors in $v.set$ are allocated to tasks in the effect of the GPRCV actions. Lemma 4.2 shows that in all complete rounds the loads of processors are balanced.

**Lemma 4.2** *[Load balancing] In algorithm AF, for each view $v$, in each round $Rnd > 0$, whenever processor $i$ is assigned to a task in the effects of the* GPRCV *action (1) for any processor $j$ that is assigned to a task in the same round, $U_i = U_j$, and (2) no more than $\lceil |v.set|/U_i \rceil$ processors are allocated to any task.*

**Proof.** Part (1) follows from the definition of $U$ and by observing that in the action GPRCV a processor is allocated to a tasks only upon receiving messages from all members of $v$. Part (2) then follows from the load balancing rule. □

**Lemma 4.3** *In algorithm AF, any processor is a member of at most $f+1$ views during the computation.*

[2]In the fragmentation failure model this scenario does not occur for the virtually synchronous group communication services. We give this lemma here in the form suited for weaker group communication services for generality.

**Proof.** Each fragmentation failure causes each processor to become a member of at most one new view. (It is possible for all processors in an existing view to successfully terminate, even if the group communication service installs subsequent views.) There are at most $f$ such views. Additionally, all processors participate in the initial view, making the total number of views for each processor to be at most $f + 1$. □

We call the last round of any view, whether complete or not, the *final* round of the view.

**Lemma 4.4** *The work of algorithm AF in all zero-numbered and final rounds of all views $v$ installed during the computation is $O(n + f \cdot n)$.*

**Proof.** By Lemma 4.3 any processor participates in at most $f+1$ views during the computation. Since there are $n$ processors, the work in the zero-numbered and final rounds is $O(n + f \cdot n)$. □

**Lemma 4.5** *In algorithm AF, in each view $v$ there can be at most one non-final completed round such that if a processor $i$ is assigned to tasks in the effects of the* GPRCV *action, then $U_i < |v.set|$.*

**Proof.** Assume round $r$ is the first such completed round. If $U_i = 0$ then all processors terminate (i.e., set *Phase* to *stop*) and there are no subsequent rounds in the view, thus round $r$ is the final round in $v$. Else if $U_i \neq 0$ and $U_i < |v.set|$ then all remaining tasks are performed in this round. If the round $r + 1$ is completed, then it is the final round in $v$. If the round $r + 1$ is not completed, then it is also the final round in $v$. □

**Lemma 4.6** *In algorithm AF, the total work in all views $v$ during non-final completed rounds with $U_i < |v.set|$ is $O(n + f \cdot n)$.*

**Proof.** Since each view has at most one such round (Lemma 4.5), and at most $n$ processors complete such rounds in any view, and since each processor is a member of at most $f + 1$ views, the total work is $O(n + f \cdot n)$. □

We call a view in which processors terminate, i.e., set *Phase* to *stop*, a *terminal* view.

**Lemma 4.7** *In algorithm AF, the total work in all views $v$ during completed rounds $r$ with $U_i^{(r)} \geq |v.set|$ is $O(n + f \cdot n)$.*

**Proof.** Consider a view $v$. By Lemma 4.2, each processor is assigned to at most one task in any completed round in this view. Let $U_v$ be any of the $U_i$ computed in the first such completed round. Then there can be no more than $\lfloor U_v/|v.set| \rfloor$ rounds $r$ with $U_i^{(r)} \geq |v.set|$. The work in such rounds in any view $v$ is thus $O(\lfloor U_v/|v.set| \rfloor \cdot |v.set|) = O(U_v) = O(n)$. Since there are at most $f + 1$ different views, the total work is $O(n) \cdot (f + 1) = O(n + f \cdot n)$. □

Now the main complexity result and its tightness.

**Theorem 4.8** *The termination work of the algorithm is $O(n + f \cdot n)$.*

**Proof.** Follows directly from Lemmas 4.4, 4.6 and 4.7. □

**Theorem 4.9** *The termination work of the algorithm is* $\Omega(n + f \cdot n)$.

**Proof.** For any fragmentation pattern with $f$ fragments that immediately follows the initial view, there are $f+1$ groups. The processors in each of these groups perform no fewer than $n$ tasks. □

It is also interesting to note that there are small dynamic fragmentation patterns that leave almost all processors connected in a large group that nevertheless accounts for most work.

**Theorem 4.10** *There is a fragmentation pattern with* $f = \log n / \log\log n$ *such that the largest group has at least* $n - n/\log\log n = \Theta(n)$ *processors at all times and has termination work of* $\Omega(n \log n / \log\log n)$.

**Proof.** Follows from a straightforward adaption of the lower bound for the analogous problem in the shared memory model as given in Theorem 4.2.4 in [15].

The adversary that leads to this result is specified as follows. In the largest current group, the adversary precomputes the pattern of assignments of processors to tasks in the group. It then allows the processors to perform the tasks and subsequently splits off into a separate group the processors that were assigned to some arbitrary $u/\log n$ tasks, where $u$ is the number of remaining tasks (for the initial group, $u$ is $n$). In [15] it is shown that this strategy can be continued for $\log n/\log\log n$ steps, thus causing $f = \log n/\log\log n$ fragmentations. Under these circumstances, the largest group still contains at least $n - n/\log\log n = \Theta(n)$ processors. Thus the work performed by the processors that complete all tasks in the largest group is $\Omega(n \log n/\log\log n)$. □

# 5 Low Redundancy Task Scheduling

In this section we consider a fully dynamic network where both fragmentations and merges are possible. Our goal is to produce a scheduling strategy that avoids redundant task executions in scenarios where there are periods in which processors work in isolation and then are merged into larger groups. In particular, we seek solutions where the isolated processors can execute tasks independently for as long as possible such that when any two processors are merged into a larger group, the number of tasks they have *both* executed is as small as possible.

**Definition 5.1** *For a set of* $p$ *processors with identifiers* $\{P_1, \ldots, P_p\}$ *and a set of* $n$ *tasks* $\{T_1, \ldots, T_n\}$, *where* $p \leq n$, *a scheduling scheme* $\mathcal{S}$ *is called* $[\alpha, \beta]$-redundant *if it provides each processor* $P_i$ *with a sequence of* $\alpha$ *tasks* $s_i = T_{i_1}, \ldots, T_{i_\alpha}$ *such that for any* $s_i$ *and* $s_j$ $(i \neq j)$, $|\{q : T_q \text{ in } s_i\} \cap \{r : T_r \text{ in } s_j\}| \leq \beta$.

It is easy to avoid redundant task executions among the tasks performed *first* by any processor. One possibility is to begin with a step in which each processor, $P_i$, executes the $i$th task. The first step does not introduce any overlaps in task execution. Clearly, in the second step we cannot avoid executing tasks that have been already executed. This means that there will always be pairs of processors such that at least one task would have been executed by both. Surprisingly, it is possible to make task scheduling decisions for a substantial number of steps such for *any pair* of processors, there is *at most one task* that is executed by both.

We start with a simple scheduling strategy that extends the simple scheduling step we described above. In this scheme, a processor $P_i$ is using a schedule $s_i = T_{i_1}, \ldots, T_{i_j}, \ldots$, where $T_{i_j}$ is the task number $i_j = (k_j + i) \bmod n$. Thus, the scheme is fully defined by the values of $k_j$, where $1 \leq j \leq n$. Note that we already fixed $k_1$ to be zero.

Next we suggest a way to determine the values of $k_j$, $2 \leq j \leq n$.

The first scheme we present, called the *logarithmic scheme*, guarantees that there is at most one overlap. This scheme uses $k_j = 2^{j-1} - 1$ for every $j$ such that $2 \leq j \leq \lfloor \log n \rfloor$.

**Theorem 5.2** *The logarithmic scheme is* $[\Theta(\log n), 1]$-redundant.

**Proof.** Assume by way of contradiction that two processors $P_x$ and $P_y$ execute two tasks $T_u$ and $T_v$. Assume, without loss of generality, that $y > x$. Note that by symmetry arguments we may assume that $y - x < n/2$ and that $P_x$ is $P_1$. The indices, $u$ and $v$, of the tasks $T_u$ and $T_v$ are chosen by $P_x$ and $P_y$, thus $u = x + 2^{z_1} = y + 2^{z_2}$ and $v = x + 2^{z_3} = y + 2^{z_4}$. Since $u \neq v$ and $x \neq y$, all $z_m$ $(1 \leq m \leq 4)$ are distinct. Thus, $y - x = 2^{z_1} - 2^{z_2} = 2^{z_3} - 2^{z_4}$ which is impossible. $\Box$

It turn out that the number of tasks that can be executed while at most one task execution overlaps is greater than $\Theta(\log n)$. In Figure 3 we present a scheme, called the *cubic root* scheme, that provides schedules of $\Theta(n^{1/3})$ tasks for the processors with only one overlap. An important observation used for the design of our algorithm is the following observation: to guarantee at most one overlap, the difference between every $k_x$ and $k_y$ must be distinct.

**Definitions:**

Let $\mathcal{K}_j$ be a set of $j$ indices $k_1, k_2, \cdots, k_j$ (that were chosen so far). Let $\mathcal{D}_j$ be the set consisting of two integers, $d[k_l, k_m]$ and $d[k_m, k_l]$ for each possible pair of elements, $k_m$ and $k_l$ in $\mathcal{K}_j$, where $d[k_x, k_y]$ is defined to be $(k_y - k_x)$ *mod* $n$.

**Initialization:**

In the beginning $\mathcal{K}_1$ includes only the element 0 and $\mathcal{D}_1$ is assigned the empty set.

**Step, calculating** $k_j$**:**

The algorithm chooses $k_j < n/2$ to be the smallest value such that $k_j$ paired with any element $k_y$ of $\mathcal{K}_{j-1}$ does not introduce an element $d[k_j, k_y] \in \mathcal{D}_{j-1}$ or $d[k_y, k_j] \in \mathcal{D}_{j-1}$.

**Termination Condition:**

No $k_j$ is found in **Step**.

Figure 3: Scheduling Tasks with One Overlap.

**Theorem 5.3** *If and only if the difference between every $k_x$ and $k_y$ is distinct then the number of overlaps is at most one.*

**Proof.** Assume there are $k_x$, $k_y$ and $k_u$, $k_v$ (where $k_y$ may be equal to $k_u$, $k_x \neq k_y$, $k_y \neq k_v$, and $k_x \neq k_u$) such that $k_y - k_x = k_v - k_u$ (and thus $k_v - k_y = k_u - k_x$). Consider processors $P_j$ and $P_{j+(k_v-k_u)}$. According to the *cubic root* scheme, at some step $t_1$ processor $P_j$ performs the task number $(k_{t_1} + j) \bmod n$, and at some step $t_2$ processor $P_{j+(k_v-k_y)}$ performs the task number $(k_{t_2} + j + (k_v - k_y)) \bmod n$. First, choosing $t_1 = u$, processor $P_j$ performs the task number $(k_u + j)$ mod $n$. Choosing $t_2 = x$, processor $P_{j+(k_v-k_y)}$ performs the task number $(k_x + j + (k_v - k_y))$

mod $n = (k_x + j + k_u - k_x)$ mod $n = (k_u + j)$ mod $n$, i.e., the same task as $P_j$ at step $u$. Next, choosing $t_1 = v$, processor $P_j$ performs the task number $(k_v + j)$ mod $n$. Choosing $t_2 = y$, processor $P_{j+(k_v-k_y)}$ performs the task number $(k_y + j + (k_v - k_y))$ mod $n = (k_v + j)$ mod $n$, i.e., the same task as $P_j$ at step $v$ (and distinct from the task $(k_u + j)$ mod $n$).

On the other hand if two processors $P_j$ and $P_{j+z}$ have two identical tasks to execute then it holds that $k_y - k_x = (k_y + z) - (k_x + z) = k_v - k_u$. ///is this sufficient or do we need more detail?/// $\square$

In general, to guarantee at most $l$ overlaps the number of pairs, $k_x$, $k_y$, with the same difference should be no more than $l - 1$.

**Theorem 5.4** *The cubic root scheme is* $[\Theta(n^{1/3}), 1]$-redundant.

**Proof.** Let $\mathcal{U}_i$ be the set of possible $k_j$ values that are candidates to be chosen while obeying the one overlap requirement after $k_1, k_2, \cdots, k_i$ are chosen. Initially $\mathcal{U}_0$ is the set of values $\{0, 1, \cdots, n-1\}$. After scheduling the first task (the one determined by $k_1 = 0$), 0 is not included in $\mathcal{U}_1$ since it was used. After choosing $k_2 = 1$, the values in $\mathcal{U}_2$ are $3 \cdots n-2$. The index 1 is excluded from $\mathcal{U}_1$ because it is used. The indices 2 and $n-1$ are excluded because they are at a distance one (the distance one is in $\mathcal{D}_2$) from a member of $\mathcal{K}_2$ (See Figure 2 for the definitions of $\mathcal{D}_2$ and $\mathcal{K}_2$). We can see that the number of members in $\mathcal{U}_j$ can be computed as a function of $\mathcal{K}_{j-1}$ as follows: When an element $k_j$ is added to $\mathcal{K}_{j-1}$ the number of elements in $\mathcal{U}_{j-1}$ is decremented by at most $1 + 2 + \cdots + |\mathcal{K}_{j-1}| + 1 + |\mathcal{K}_{j-1}|$: The first member that is removed from $\mathcal{U}_0$ is $k_j + (k_2 - k_1)$, the second and third elements are $k_j + (k_3 - k_1)$ and $k_j + (k_3 - k_2)$. The fourth, fifth and sixth elements that are removed from $\mathcal{U}_{j-1}$ are $k_j + (k_4 - k_1)$, $k_j + (k_4 - k_2)$ and $k_j + (k_4 - k_3)$, continuing the same way we obtain the sum $1 + 2 + \cdots + |\mathcal{K}_{j-1}|$. The next additional 1 is due to $k_j$ itself and the last $|\mathcal{K}_{j-1}|$ is due to the distances from $k_1$ in the opposite direction: $(1 - d[k_j, k_1])$ mod $n$, $(1 - d[k_j, k_2])$ mod $n$, and so on. Hence, the number of members that are removed from $\mathcal{U}_{j-1}$ following the choice of $k_j$ is $1 + 2 + \cdots + j - 1 + 1 + j - 1 = (j+1)j/2$. The total number of elements that are removed from $\mathcal{U}_0$ is $\Sigma_{l=1}^{l=j}(l+1)l/2 < j^3$. The proof is completed by using Theorem 5.3 and the fact that in our construction every two elements in $\mathcal{K}_j$ are in distinct distance. $\square$

The schemes presented in this section allow the processors to schedule tasks in isolation. This is the case when the processors find themselves in singleton groups. We now suggest a way to use the scheme when groups are merged or when larger groups are formed initially. Processors within a group identify the overlapping task executions and agree which is the single processor within the group that executes each such task. The processors will continue to execute the tasks in their ("singleton") schedule that are not executed by other processors in the group. Thus, in case the system is partitioned into singletons, at most one overlap between every two processors is achieved for $\Theta(n^{1/3})$ steps and still no redundant task execution exists within a group.

# 6 Concluding Remarks

We considered the problem of dynamic load balancing in networks subject to reconfigurations, and we have presented three new directions in the investigation of load balancing with group communication. First, we have shown that in the presence of fully dynamic changes no on-line algorithm can do much better than the trivial solution in which every processor behaves as if it is a singleton and executes all tasks. This led us to examine the last two scenarios. For fragmentation failures we presented an algorithm that guarantees completion with total work $O(n + f \cdot n)$, where

$f$ is the number of fragmentation failures. Finally, for the case of fully dynamic reconfigurations we presented a scheduling strategy for minimizing the task execution redundancy between processors that can schedule $\Theta(n^{1/3})$ tasks with at most one overlap of task execution for any two processors. Finally, the problem of minimizing overlaps for singleton groups bears similarity to some problems in design theory and we intend to pursue this connection in future work.

**Acknowledgments:** We thank Nancy Lynch and Dahlia Malki for several discussions that motivated parts of this work. We also thank Shmuel Zaks for insightful remarks and Chryssis Georgiou for helpful comments.

# References

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

[2] O. Babaoglu, R. Davoli and A. Montresor. "Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specification," in *Operating Sys. Review*, 31(2):11-22, April 1997.

[3] O. Babaoglu, R. Davoli, A. Montresor and R. Segala, "System support for partition-aware network applications", in *Proc. of the 18th Int-l Conference on Distributed Computing Systems*, May 1998.

[4] M. Hayden, doctoral thesis, Cornell University, 1999.

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

[6] B. Chlebus, R. De Prisco and A. Shvartsman, "Performing tasks on restartable message-passing processors", in *Proc. of the 11th Int-l Workshop on Distr. Alg. (WDAG'97)*, pp. 99–114, 1997.

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

[8] 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.

[9] 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.

[10] C. Dwork, J. Halpern, O. Waarts, "Performing Work Efficiently in the Presence of Faults", *SIAM J. on Computing*, 1994; prelim. vers. appeared as Accomplishing Work in the Presence of Failures in *Proc. 11th ACM Symposium on Principles of Distributed Computing*, pp. 91-102, 1992.

[11] 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*, 1995, pp 296–306.

[12] A. Fekete, R. Khazan 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.

[13] A. Fekete, N. Lynch, and A. Shvartsman, "Specifying and Using a Partitionable Group Communication Service," *Proc. of the 16th Annual ACM Symp. on Principles of Distributed Computing*, pp. 53-62, 1997.

[14] 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*, 1995, pp. 724–733.

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

[16] 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.

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

[18] 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.

[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] 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.

[21] 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.

[22] D. Sleator and R. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *CACM 28*, pp. 202-208, 1985.

[23] 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.

## Appendix: VS Service

The functions provided by the VS service [13], given in Figure 4, are sufficient for algorithm $AF$. The algorithm does not assume that messages are ordered within a view and does not take advantage of the SAFE notifications.

---

**Data types:**

$\mathcal{M}$ : messages
$\mathcal{P}$ : processor ids
$\mathcal{G}$ : group ids
$views = \mathcal{G} \times \mathcal{P}$ : group views with selectors $id$ and $set$

$m \in \mathcal{M}$
$p, q \in \mathcal{P}$
$g_0, g \in \mathcal{G}$ ($g_0$ is a distinguished view)
$v \in views$

**States:**

$created \subseteq views$, initially $\{\langle g_0, \mathcal{P}\rangle\}$
for each $p \in \mathcal{P}$:
  $current\text{-}viewid[p] \in \mathcal{G}$, initially $g_0$
for each $g \in \mathcal{G}$:
  $queue[g]$, a finite sequence of $\mathcal{M} \times \mathcal{P}$, initially $\emptyset$

for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
  $pending[p, g]$, a finite sequence of $\mathcal{M}$, initially $\emptyset$
  $next[p, g] \in \mathbf{N}^{>0}$, initially 1
  $next\text{-}safe[p, g] \in \mathbf{N}^{>0}$, initially 1

**Transitions at $i$:**

**output** CREATEVIEW$(v)$
Precondition:
  $v.id > \max\{g : \exists S : \langle g, S\rangle \in created\}$
Effect:
  $created \leftarrow created \cup \{v\}$

**output** NEWVIEW$(v)_p$
Precondition:
  $v \in created$
  $p \in v.set$
  $v.id > current\text{-}viewid[p]$
Effect:
  $current\text{-}viewid[p] \leftarrow v.id$

**input** GPSND$(m)_p$
Effect:
  append $m$ to $pending[p, current\text{-}viewid[p]]$

**internal** VS-ORDER$(m, p, g)$
Precondition:
  $m$ is head of $pending[p, g]$
Effect:
  remove head of $pending[p, g]$
  append $\langle m, p\rangle$ to $queue[g]$

**output** GPRCV$(m)_{p,q}$, hidden $g$
Precondition:
  $g = current\text{-}viewid[q]$
  $queue[g](next[q, g]) = \langle m, p\rangle$
Effect:
  $next[q, g] \leftarrow next[q, g] + 1$

**output** SAFE$(m)_{p,q}$, hidden $g, S$
Precondition:
  $g = current\text{-}viewid[q]$
  $\langle g, S\rangle \in created$
  $queue[g](next\text{-}safe[q, g]) = \langle m, p\rangle$
  for all $r \in S$:
    $next[r, g] > next\text{-}safe[q, g]$
Effect:
  $next\text{-}safe[q, g] \leftarrow next\text{-}safe[q, g] + 1$

---

Figure 4: View synchronous group communication service