Group Membership and Wide-Area Master-Worker Computations

Kjetil Jacobsen* kjetilja@cs.uit.no
Xianan Zhang† xzhang@cs.ucsd.edu
Keith Marzullo† marzullo@cs.ucsd.edu

Abstract

Group communications systems have been designed to provide an infrastructure for fault-tolerance in distributed systems, including wide-area systems. In our work on master-worker computation for GriPhyN, which is a large project in the area of the computational grid, we asked the question *should we build our wide-area master-worker computation using wide-area group communications?* This paper explains why we decided doing so was not a good idea.

Keywords: master-worker, group membership, network partitions, redundant tasks, omni-do.

1 Introduction

The GriPhyN[1] (**Gri**d **Phy**sics **N**etwork) is a NSF-funded research project to implement a Petabyte-scale computational environments for data intensive research projects. The project, whose requirements are being defined in the context of four current large physics experiments, will deploy computational environments called Petascale Virtual Data Grids (PVDGs) that will meet the data-intensive computational needs of a diverse community of thousands of scientists spread across the globe.

One of the functions of a PVDG will be the reconstruction of "virtual" data, which is data that is derived from the raw or processed output of experiments. Reconstruction is highly parallelizable, and so we use a master-worker-like computation for reconstruction. Some reconstructions will be large enough that we believe that we will want to execute them utilizing the resources of several computation farms spread across the Internet. Hence, we are looking into the issues of wide-area

*Department of Computer Science, University of Tromsø, N-9037 Tromsø, Norway. Supported by NSF (Norway) grant No. 112578/431 (DITS program).

†Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114, USA. Supported by NSF ACI-0086044.

[1]GriPhyN homepage: http://www.griphyn.org/

master-worker computations. In particular, the authors of this paper, who are members of the GriPhyN collaboration, are looking into the scalability, fault-tolerance, and performance problems of wide-area master-worker computations.

Group communications systems are designed to provide an infrastructure for fault-tolerance in distributed systems, including wide-area systems. We give a brief overview of the support they provide in the next section. Master-worker has been advocated at ICDCS as an application that fits well with partitionable group communications [6] and has been formally studied in the context of group communications [12]. Furthermore, there are partitionable group membership protocols that have been designed explicitly for scalability, including one that was developed by a group that included an author of this paper [14]. Hence, an obvious question to us was *should we build our wide-area master-worker computation using group communications?* To answer this question, we chose as our objective a reasonable metric, the number of redundantly executed tasks, that has been studied in the context of group communications-based solutions. We first examine a protocol that is optimal with respect to this metric, and observe how group communications is used to come up with an optimal solution. We then show, via trace analysis, that this protocol performs poorly with respect to another important metric, total completion time. The reason for this poor performance is in scalability: the larger the network, the more likely communications will not be transitive and symmetric. One can address this issue, but doing so can create other problems. We then show that a simple solution, which has a much larger worst-case complexity in the number of redundant tasks, appears to work much better in practice. Hence, we have decided that partitionable group communications is a poor choice for building scalable wide-area master-worker computations.

2 System Model

We assume a distributed system comprised of processors connected to local area networks, where the local-area networks are connected into a wide-area network. Processes on different processors can only communicate by sending messages over the network. We assume that processors can fail by crashing and that the network can fail by dropping messages. We assume that communications provide FIFO delivery order: if process p sends message m to process p' and then sends message m' to process p', then p' may receive just m, or just m', or m before m', but never m' before m.

We assume that the system is timed asynchronous [10]: processors have clocks that progress at a rate close to real-time, but the time between some process p sending a message and the intended destination p' delivering the message can be arbitrarily long.

In Section 3 we assume that there exists a partitionable group communication system. *Group communication* [1, 8] provides reliable multicast communication among processes that are organized into groups. A *group* is a set of processes that together comprise the *members* of the group. A process becomes a group member by requesting to *join* the group; it can cease being a member by requesting to *leave* the group or by failing.

Each group is associated with a name. Processes multicast to group members by sending a message to the group name, and the group communication service delivers the message to the group members (depending on the ability of the members to communicate, which we elaborate on below).

Group communication systems are *view oriented*, which means that they provide membership information and deliver messages in a well-defined order among all members. Such systems provide a useful abstraction for the development of highly available distributed and communication-oriented applications. Group communications systems differ in the details on how they implement such an ordering.

The *membership* of a group is a list of the currently active and connected processes in a group. The task of a *group membership service* is to track the membership of the group as it evolves over time. When the membership changes, the application processes are notified at an appropriate point in the delivery sequence. The output of the membership service is called a *view*, which consists of the list of the current members in the group and a unique identifier that allows the application to distinguish the view from other views with the same list of members. Views are used in two main ways:

1. Since the members of the view agree on the membership of the view, they can deterministically assign roles to each other without using further communication. For example, the first process in the membership list can serve as a coordinator for all the processes in the membership list.

2. Consider any two processes that are both in the membership of a view v when they both install the same new view v'. The group communications system ensure that both processes

have delivered the same sequence of messages while in view v.

A *partitionable group membership service* (for example, [4, 14]) is a group membership service that is designed to operate in wide-area networks and that supports writing *partition-aware* applications, which are applications that can continue to make progress despite network partitions [5]. A *network partition* is the situation in which two sets of nonfaulty processes cannot communicate with each other due to problems at the network layer or lower. We say that two processors in which at least one processor can communicate with each other are in the same *connected component* of the network. Hence, a partitioned network has more than one connected component.

Partitionable group membership protocols monitor the network connectivity using an underlying unreliable failure detection service. The failures it reports are used to instigate view changes. When the failure detection service stabilizes such that communication is possible among all the processors in a connected component, a new view can be delivered to the processes that are running on processors in the connected component. Different partitionable group communications systems have different delivery rules associated with messages that are sent while the failure detection service is stabilizing; [20] contains a summary of the rules of six different protocols. Some systems allow messages to be multicast during such periods of unstable failure detection, while others do not allow multicasting. In the systems in which messages can be multicast, such messages are not delivered until the next view is installed. Hence, some blocking of communications occurs during the time that a connected component's failure detection relation lacks symmetry or transitivity (and so is not a clique). Such blocking could be avoided by implementing the missing transitivity and symmetry by routing within the group communications system (one system, Phoenix, did exactly this [19]) but doing so creates other problems, as we discuss in Section 6.

3 A Protocol that uses Group Membership

There has been a vast amount of work that has been done on the problem of master-worker computation. Only a small amount of it has been into the question of protocols for master-worker computation in a wide-area network that can suffer from partition failures. The problem was recently identified as one amenable to *partition aware solutions* [6], which are applications that can continue to make (perhaps limited) progress during the period of time that a wide-area network

is partitioned. It has been argued that group communications services for wide-area networks provide a convenient framework for writing such applications, although the exact details of group communication may have a significant impact on the design (for example, see [20] and [5]).

There has been recent work into the complexity of solving a variation of master-worker. This work considers the case in which the amount of redundant task execution is to be kept as small as possible (e.g., [17, 11, 18]). This research project has produced a protocol, named *AX*, that is based on group communication services [12]. The variation of master-worker that it solves is called the *OMNI-DO* problem:

OMNI-DO Problem 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.

This variation is important from a theoretical viewpoint because a centralized version (in which only the master need know the results of all the tasks) can minimize the amount of redundant task execution by using stable storage and a simple failover mechanism for recovering from a failed master. Workers that can partition away from the master would not compute any tasks, since the results could be unavailable from the master for an unbounded time, and so termination can require redundant task execution. From a practical point of view, though, it is also important for the computation to complete in a timely manner, which would drive one to use a partition-aware approach. Furthermore, a practical wide-area master-worker computation, like OMNI-DO, would distribute the results among several local area networks so that the user would be able to obtain the results even in the face of a long-lived partition.

AX employs a coordinator-based approach and relies on the underlying partitionable group membership service to track changes in the group's composition. There is a set of processes, one for each processor in P, that are cooperating to solve the OMNI-DO problem, and there is a set of tasks T known by all of the processes that are to be computed. All tasks have the same duration.

During execution, each process i maintains the local sets D_i, R_i, U_i and G_i:

D_i – The set of tasks whose results process i knows.

R_i – The results of the tasks in D_i.

U_i – The set of tasks whose results process i does not know: $U_i = T - D_i$.

G_i – The set of processes in i's view.

For each process i, $rank(i, G_i)$ is the rank of i in G_i, when the process identifiers are in some well-known order, such as the order they appear in the view membership list. For a task u in U_i, $rank(u, U_i)$ is the rank of u in U_i, when the task identifiers are sorted in ascending order.

The task allocation rule for each processor i is:

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

AX structures its computation in terms of rounds. Each process executes at most one task in each round. At the beginning of each round, each processor i knows G_i, D_i, U_i and R_i. Since all processors know G_i, each processor deterministically chooses a group coordinator, which is the process with the highest ID in G_i. In each round, each processor i reports D_i and R_i to its coordinator. The coordinator receives and collates these reports and broadcasts the results to the members of the group. After receiving this broadcast message from the coordinator, each process i updates D_i, R_i, and U_i, and then chooses a new task to compute using the task allocation rule.

Initially, all processes are members of a single initial view that contains all the processes. If a regrouping occurs, then the affected processes receive the new views from the group membership service, complete any tasks that they are currently computing, and report the results to the new coordinators. Each new coordinator will then start the first round in the new view.

We can classify all the tasks into three types as follows. Given a view G:

Fully done tasks $FD(G)$: the tasks $\{t \in T | \forall i \in G : t \in D_i\}$.

Partially done task $PD(G)$: the tasks $\{t \in T | \exists i, j \in G : t \in D_i \wedge t \in U_j\}$.

Undone tasks $UD(G)$: the tasks $\{t \in T | \forall i \in G : t \in U_i\}$.

Consider the situation in which the view G partitions into two or more views. Clearly, all the tasks in $FD(G)$ won't be re-executed by any process in G. In any algorithm, all of the tasks in $UD(G)$ are at risk of being executed redundantly because of the need for liveness: the partition may last for an unbounded time, and so the tasks in $UD(G)$ may need to be computed by at least

one process in each new view. Any task u in $PD(G)$ is also at risk of being executed redundantly in any new view that does not know u's result. By using a round structure, AX ensures that the size of $PD(G)$ is never larger than $|G| - 1$. No algorithm can ensure $PD(G)$ is smaller unless it does not allow all processes in G to be computing at the same time. Hence, using a round structure is one way to reduce the number of redundant tasks executed.

There are other ways to ensure $|PD(G)| < |G|$; for example, each process i could broadcast its result of executing task u to G_i rather than just sending it to the coordinator. Process i would not allocate another task until it knew that all other processes j in G had updated D_j, R_j and U_j. AX was not designed this way because it would introduce many more messages [12], and given that the tasks in T all have the same execution time, there would not be much to be gained if this additional communication were used.[2]

Even though it is optimal (in terms of the worst-case number of tasks executed, as a function of the number of views installed) and has a low message overhead, AX was not meant to be a practical algorithm. It has some obvious problems that can be easily addressed. For example, each processor sends D_i and R_i to its coordinator at the end of each round. Doing so makes the propagation of results when views merge trivial to implement, but it could result in a huge message overhead. It would be easy for i to limit the size of D_i and R_i by having each process maintain a vector denoting the results it knows. Such a vector could probably be kept small with a suitable encoding technique. These vectors could be managed in a manner similar to logical clocks [15] so that when i sends a result to its coordinator, it only sends the part of D_i and R_i that i does not know that the coordinator knows.

Another problem that could be easily addressed would reduce the number of tasks executed redundantly in some runs (recall that AX is optimal only in terms of the worst-case behavior). Consider what AX does if the processes partition into two or more views in the initial state. The processes in each view will deterministically start executing the same set of tasks. One could instead have a coordinator choose tasks randomly without replacement for the processes in its view to compute. If T is large and the partition does not last a long time (as measured in task computation time), then this would result in a smaller expected number of redundant task execution.

[2]Note that the problem can be solve with *no* message communication, and so trying to include optimality in terms of message complexity is not an interesting exercise. [17]

A third problem arises from its use of coordinators. In a real system the tasks would not have exactly the same execution time. Each round runs as long as the longest task in that round, and with a large number of workers, the longest task in each round could be quite long. Despite the larger number of messages, the variation of AX we gave above that does not use a coordinator would probably be a better choice.

In the next section, we discuss a more important practical problem with AX which arises from its use of group communications. This problem occurs even if all the changes above are made to AX.

4 Analysis of AX

AX was designed with the idea of reducing the number of redundantly computed tasks. Redundant task execution can occur when the network partitions. The exact number of redundant tasks executed depends on many factors, including the length of time a task executes, the number of processes, how the processes partition from each other, at what point the partitions occur, and for how long each endures.

AX was not explicitly designed to compute the results of OMNI-DO quickly, but in fact it would appear to do so. It is work conserving: if there is a task to be executed and a process available to execute a task, then the task is executed. Since tasks have the same computation time, blocking due to the round structure should be small and the order that the tasks are executed is immaterial. Hence, since AX reduces the number of redundant tasks, it would appear to be a fast algorithm as well. However, as was discussed in Section 2, group communication services can perform poorly when communication is not symmetric and transitive. The actual performance of AX depends strongly on the frequency and duration of nonsymmetric or nontransitive communication.

To understand the performance of AX in practice, we need to understand how communication breaks down: how symmetric and transitive communication is and how often a network partitions. Furthermore, worst-case behavior is not necessarily the most important metric to consider. In practice, the average execution time is probably of more interest to users of the system.

Unfortunately, there are no accurate existing models of network connectivity that would allow us to analyze AX. Hence, we follow other similar work (e.g., [2] and [7]) and resort to using network

traces. Using trace data means that it is difficult to generalize from our findings to other network configurations. Hence, we also give a simple model for the expected behavior of AX given metrics that can be measured. We show via simulation that the model has predictive value.

4.1 Trace data

We used two different sources for trace data sets. The first data set comes from the Resilient Overlay Networks (RON) project [2]. RON is an application-layer overlay on top of the existing Internet routing substrate. The nodes that comprise RON monitor the function and quality of the Internet paths among themselves, and use this information to decide whether to route packets directly over the Internet or by way of other RON nodes. The RON project uses a testbed deployed at sites scattered across the Internet to demonstrate the benefits of the architecture. Since the probing reported in the trace data referred to in [2] is not frequent enough for our analysis, we used another trace that was kindly collected and supplied to us by the RON group [3].

In this trace, there are sixteen nodes that are spread across the United States and Europe. Each pair of nodes probe each other via UDP packets once every 22.5 seconds on average. To probe, each RON node independently picks a random node j, sends a packet to j, records this fact in a log, records if there was a response, and then waits for a random time interval between one and two seconds. The logs from each machine are then collected and merged into a single trace.

From this trace, we generated a directed *communications graph* with the RON nodes as the vertices in the graph. Ideally, an edge is drawn from node i to node j if a process at j can successfully receive messages from a process in i. A node i may be marked as *crashed*, which indicates that the process at node i is crashed.

The graph is initially connected and all nodes are marked as not crashed. A node i is marked as crashed if the log indicates that i did not send a probe for five minutes. Once i subsequently sends a probe, the node i is marked as not crashed. If the trace shows that the last three messages (either probes or responses) a node i sent to j were not received by j, then the directed edge from i to j is erased. The edge from i to j is added again when the trace records j having received a packet from i (either a probe or a response). The trace contains continuous probing for the two week period from August 2 through August 16, 2002.

Group communications services for wide-area networks can be factored to run on top of specialized failure detector services [14], and so it can be hard to come up with a general connectivity model that would predict how group communications services would behave in any given situation. One obvious model, however, is based on TCP connectivity. UDP connectivity is clearly worse than TCP connectivity since packets can be lost due to congestion and there is no retry (unlike TCP). We chose the method of declaring a link down only when three packets are lost to make the connectivity graph a more conservative estimator of how group communications services would perform.

The second set was collected by Omar Bakr and Idit Keidar. They studied the running time of TCP-based distributed algorithms deployed in a widely distributed setting over the Internet. They experimented with four algorithms that implement a simple and typical pattern of message exchange that corresponds to a communication round in which every host sends information to every other host. The traces we analyzed are described in [7].

The information was collected from universities and commercial ISP hosts. The machines were spread across the United States, Europe, and Asia. Each host sent a `ping` (ICMP) packet to each other host once a minute. Each process records to a local log the ping packets it receives from other processes.

The communications graph is constructed in a similar way as was done for the RON traces. The graph is initially connected. The edge from a process i to another process j is removed when three minutes elapses without j having received a ping from i. The edge is put back when j finally receives a ping from i. There is not enough information in these traces for us to be able to mark a node as being crashed, and no node is ever marked as being crashed.

There were three traces generated in total: one with nine nodes, one with eight nodes, and one with ten nodes. We denote these three traces as *exp1*, *exp2* and *exp3*. Each trace recorded more than three days' worth of probing. In exp1, two links had high loss rates: one from National Taiwan University to a commercial site in Utah sustained a 37% loss rate, and another from National Taiwan University to a commercial site in California sustained a 42% loss rate. In exp3, links from National Taiwan University and from Cornell University had high loss rates. Because of these high loss rates, in [7] they also considered the subset of exp1 with the process at National

RON (16 nodes): F=18%, Pct Partition time = 3.9%				
	Non-partition period	Partition period		
		Start time	Duration	Two-clique time
1st	60,843.5	60,843.5	2.4	100%
2nd	259,966.6	320,812.5	1.7	100%
3rd	290,407.9	611,222.1	56.4	100%
4th	41,421.3	652,699.8	27,639.6	98%
5th	224,965.6	905,365	70.6	0%
6th	16,591.4	922,027	27.9	0%
7th	149,180.1	1,071,235	98.2	100%
8th	7,995.6	1,079,328.8	22.4	100%
9th	912	10,80263.2	95.9	100%
10th	60,003.8	114,0362.9	92	100%
11th	49,995.9	1,190,450.8	19149	94%

exp1 (9 nodes): F=32%, Pct Partition time = 0.89%				
	Non-partition period	Partition period		
		Start time	Duration	Two-clique time
1st	74,520	74,520	3,720	82%
2nd	21,360	99,600	300	100%
3rd	360,900	-	-	-

exp2 (8 nodes): F=5%, Pct Partition time = 0%

exp3 (10 nodes): F=46%, Pct Partition time = 0%

Table 1: Partitions in the four traces.

Taiwan University removed (they did not analyze exp3 in their paper). We did not remove the data concerning National Taiwan University and Cornell from our traces because we felt that leaving it in represented a real-world situation.

4.2 Trace analysis

First, we would like to know how often partitions happen and what duration they have. A partition occurs when the communications graph contains more than one component, and all components contain non-crashed nodes. The partition data is shown in Tables 1. Each partition is identified by when it started in the trace and how long it endured. The percentage of time during which there was a partition in each trace is reported in Table 1. Summarizing,

1. There are eleven partitions in the RON trace. Nine of the eleven endured for less than two

minutes, and the two others endured for over five hours. For over 96% of the trace there was no partition.

2. There are two partitions in the exp1 trace. One lasted for five minutes and the other lasted for over an hour. For over 99% of the trace there was no partition.

3. There were no partitions in the exp2 or exp3 traces.

Since there is no information in the three traces about crashes, the partitions time reported for exp1 is only an upper bound.

All partitions that we found in the traces resulted in exactly two connected components. In all but one of the partitions, one of the components contained exactly one node; in the case, which is in the RON trace, the smaller component contained two nodes (one of these two nodes subsequently crashed during the partition). This two-node component was symmetrically and (by definition) transitively connected.

Second, we would like to know how often communication is not symmetric and transitive. At any point in time, each process i is in a connected component of the communications graph. For a process i, let $c(i)$ be the fraction of time during a trace in which its connected component is a clique. Then we compute:

$$F = 1 - \sum_{i \in \Pi} c(i)/P$$

Larger values of F indicates that AX will be less likely to make progress because of the lack of a fully formed view. Table 1 reports the values of F for the four traces we considered. All traces record a significant value for F, and trace exp2 has the lowest value.

Recall that in all partitions, the smaller of the connected components contained either one or two nodes and was both symmetrically and transitively connected. The larger connected component, of course, was not always symmetrically and transitively connected. If it is not, then AX may be blocked in the larger component, thereby reducing the number of redundant tasks computed. So, we also list the fraction of time of each partition during which the larger component was symmetrically and transitively connected. For the most part, the larger component's communications graph is a clique.

Looking at these values, we observe that:

- The periods of time during which there is no partitioning are usually quite long. In the RON traces, the shortest such period was about 15 minutes and the longest period was over three days, and in the exp2 and exp3 traces, both of which record three days' worth of probing, there were no partitions. Thus, we would expect that in the common case and with computations that span no more than a day or so, AX should not compute many redundant tasks.

- When they happen, partitions can endure for a long time. In the RON trace, the longest partition lasted for over seven hours, and in the exp1 trace the longest partition lasted for over an hour. This implies that, as was stated in [6], OMNI-DO is an appropriate problem for a partition-aware solution; making progress during such long periods is obviously desirable.

- The periods of time during which communications are not symmetric and transitive is significant. It appears that such periods are often due to one troublesome node or link. For example, from the RON traces during partitions, the larger connected component is usually fully connected. Being able to predict which nodes will be the troublemaker, though, may be hard. For example, Cornell participated in all the traces but was troublesome only in trace exp3. Thus, we expect that AX will not be as fast as one might think because of its use of group communications.

  We can estimate the slowdown factor of AX with a simple model. Assume that the network does not partition but suffers from periodic times of poor connectivity: for α seconds it is not a clique, for β seconds it is a clique, for α seconds it is not a clique, and so on. Let each task compute for T seconds. Since the tasks have the same length, they will complete at close to the same time. The master then broadcasts the new assignment to start the next round. With probability $\alpha/(\alpha+\beta)$, the broadcast will occur during an α period, and will block on average $\alpha/2$ seconds; otherwise, the broadcast occurs during a β period and does not block. Hence, it blocks on average $\alpha^2/2(\alpha+\beta)$ seconds. The slowdown factor should then be $(T+\alpha^2/2(\alpha+\beta))/T$. This simplifies to:

$$\text{Slowdown factor} = 1 + \alpha F/2T$$

4.3 Simulation results

In order to see if our observations above are correct, we simulated AX running over the RON trace. We used the communications graph to generate a set of view changes, and we used the group membership semantics in which a process's multicast is blocked while its connected component is not a clique. Given the round nature of AX, our results would not differ if only the delivery of multicast messages were blocked during these times.

We assumed that each of the sixteen nodes monitored by RON have ten processes. We assume that since processes in one node are in the same local area network, the communication among them is always symmetric and transitive. We assume that there are 1,000 tasks that run for a time much longer than it takes for messages to be transmitted over the wide-area network, and so we assume that communications is effective instantaneous among a component when its communication graph is a clique (otherwise, it is blocked as described above). A run $Sim(i)$ indicates a simulation in which each task runs for $100i$ seconds.

We first ran the simulations over a period of time in the RON traces during which there were no node crashes and there were no partitions. The results of our simulation are shown in Table 2. The column "Start time" indicates where in the trace the simulation was started. The row "Slowdown" gives the slowdown factor for different simulations and the row "F" gives the value of F for the segment of the run during which the simulation ran.

F was either very small or very large. When F is small, there is no effect on the running time, and large values of F can make AX runs between about 50% to over 11 times slower. As predicted, the slowdown factor decreases with increasing T. The slowdown factor equation requires a value for α, and so we computed a histogram of values of α and β over the entire run (when the network was partitioned we considered only the larger connected component). Table 3 gives the histogram. The average and median values for α are 143 and 22 seconds, and for β are 665 seconds and 162 seconds. The ranges of values for both are very large. If we use the average value of α in the slowdown formula, then the slowdown factors in Table 2 are usually much higher than the slowdown formula predicts. We believe that this is because some larger periods of nontransitive communication happened during the simulation.

We then ran $Sim(1)$ over five different time intervals in the RON trace during which F was

Start time	Value	$Sim(1)$	$Sim(2)$	$Sim(3)$	$Sim(4)$	$Sim(5)$
500,000	Execution time	700	1,400	2,100	2,800	3,500
	Slowdown	1	1	1	1	1
	F	0%	0%	2.7%	2.1%	1.6%
510,000	Execution time	700	1,400	2,100	2,800	3,500
	Slowdown	1	1	1	1	1
	F	0%	0%	0%	0%	0%
520,000	Execution time	700	1,400	2,100	2,800	3,500
	Slowdown	1	1	1	1	1
	F	1%	0.5%	0.9%	0.6%	0.5%
530,000	Execution time	3,266	6,320	6,320	6,858	8,156
	Slowdown	4.67	4.51	3.01	2.45	2.33
	F	99.2%	99.2%	99.2%	99.3%	99.4%
540,000	Execution time	2,696	3,780	4,081	5,211	5,211
	Slowdown	3.85	2.70	1.94	1.86	1.49
	F	98.6%	98.9%	98.7%	98.6%	98.6%
550,000	Execution time	7,819	9,573	10,188	10,188	10,825
	Slowdown	11.17	6.84	4.85	3.64	3.09
	F	99.4%	99.2%	99.1%	99.1%	99.2%
Average	Execution time	2,647	3,979	4,482	5,110	5,782
	Slowdown	3.78	2.84	2.13	1.82	1.65

Table 2: AX simulation results for traces *with no* partitions.

Time range	Non-transitive views		Transitive views	
	Number	Percentage	Number	Percentage
10	425	28.41%	240	16.05%
100	1,271	84.96%	637	42.61%
1,000	1,460	97.59%	1,227	82.07%
10,000	1,494	99.87%	1,489	99.60%
100,000	1,496	100%	1,495	100%

Table 3: The length of views.

Start time	526,900	688,000	856,500	1,070,000	1,189,300	Average
Execution time	1,118	813	1,498	1,012	976	1,083
Slowdown	1.6	1.16	2.14	1.45	1.39	1.55
F	52%	41%	55%	44%	50%	48.4%

Table 4: Simulation with $\alpha = 0.5$.

	Partn 1	Partn 2	Partn 3	Partn 4	Partn 5
Start time	652,700	905,365	922,027	1,071,235	1,140,363
Duration	27,700	71	28	98	92
Execution time	10,000	19,104	2,642	3,852	761
Slowdown	14.3	27.3	3.8	5.5	1.1
Redundant tasks	1,000	10	0	10	10

Table 5: AX simulation results for traces *with* partitions.

close to 50%. The results are shown in Table 4. With $\alpha = 143$ seconds, the slowdown formula predicts a slowdown factor of 1.36, which is close to the values computed by simulation.

Since no partitions occurred during any of these runs, the number of redundantly executed tasks should be zero, which is what the simulation reported. We then ran $Sim(1)$ starting at five different times, where at each starting time there was a partition but no crashed nodes. The results are shown in Table 5. In the first simulation, the network was partitioned throughout the whole execution, and so all tasks were executed redundantly. In the other cases, the partition lasted for only a short time. Here, since the partition result in one side having a single node with ten workers, and redundant executions starts when there is a partition, there will be only ten redundantly executed tasks. In the third simulation, the number of redundant tasks is zero because the larger component is not a clique during the partition, and so only the smaller component makes progress.

5 A Simpler Approach

AX uses group communications to reduce the number of redundant task executions during partitions, but it runs more slowly because of the more prevalent problem of poor network connectivity. By running more slowly, which is bad in itself, it can also increase the chances of redundant task execution because it is more likely to encounter a partition.

In this section, we give a simple protocol called WAMW[3], that we believe is more appropriate for the network connectivity we observed. It is a partition-aware solution, but does not require group communications. It uses leases [13] rather than an underlying failure detection service, but it would be easy to extend to use such a service if one were available.

[3]Short for **W**ide-**A**rea **M**aster-**W**orker.

Trace	Time $D(G) > 1$	Time $D(G) = 2$	Percentage
RON	215,043.4	214,376.8	99.7%
exp1	145,560	138,360	95%
exp2	235,20	22,440	95%
exp3	213,180	208,260	98%

Table 6: Maximum distance.

5.1 Link Failures

Our protocol should make progress even when there are faulty links. Flooding protocols and gossip protocols [9, 16] are often used in such circumstances. Let $d(a, b)$ be the shortest distance between nodes a and b in a graph G. The time it takes to complete a multicast in the communications graph using flooding depends on $D(G) = \max_{a,b \in G} d(a, b)$. We call $D(G)$ the *maximum distance* of G.

Table 6 gives information on $D(G)$ where G is the the communications graph when not partitioned and the larger connected component when partitioned. The first column gives the amount of time that G is not a clique, and the second column gives the amount of time that $D(G) = 2$. The third column is the percentage of time that the graph is not a clique and $D(G) = 2$. These values are all close to 1, and so a flooding protocol should be fast most of the time. The observation that $D(G)$ is rarely more than two has been noted by many others, and recently by [2].

5.2 Algorithm Structure

WAMW uses masters and workers to perform a computation. Masters schedule tasks, and workers request and execute tasks given by the masters. Each LAN has at least one master and its set of workers. Complete pseudocode for the master and the worker is in Appendix A.

A worker executes one task at a time. When a worker starts up, it requests a task from its designated master by sending a TASK_REQUEST message. Upon receiving a task, the worker starts executing it and sends the result of the task to the master upon completion. A worker terminates when its master responds with no task to be executed. A master first checks if there is a worker request pending. If so, the master extracts the result from the previous task done by the worker from the request, and marks the task as done. The worker is subsequently added to a list of idle

workers. The master then checks if there are any unallocated tasks left to be performed. If there are tasks to be performed, the master allocates a task, picks an idle worker and sends the task to the worker. The masters are terminated after every master knows the results of all tasks.

WAMW uses two types of leases. The first lease is the *worker lease*. When a worker is given a task by a master, the master registers a lease on the task. The value of the worker lease is the estimated completion time of the task. If a worker finds that it cannot complete its task within the estimated completion time, it calculates a new lease and sends a RENEW_LEASE message to the master. If, however, a master finds that a worker lease has expired, the worker is assumed to have crashed. The second type of lease is the *master lease*. Every master registers a master lease with every other master. If a master lease expires, then the master for which the lease expired is assumed to have crashed or partitioned away.

As in AX, task allocation in WAMW assumes that all masters know all tasks T to be executed in advance. We also assume that all masters know the identity of all other masters and that masters are totally ordered by $0, 1, \ldots, (n-1)$ where n is the number of masters. Initially, the complete task list for the computation is divided evenly among all masters. The size of the slice for each master is $|T|/n$. The slice of tasks to perform for master i, S_i, is then given by $S_i = T[i \times size, (i+1) \times size]$ and the *remote set* R_i for master i is defined by $R_i = \{S_0, S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_{n-1}\}$.

Task are allocated by master i to waiting workers as follows: if there is an unallocated task in S_i, then allocate the first such task; else allocate the first unallocated task in R_i. When a worker lease held by master i expires, any task it has allocated is put back into S_i or R_i. When a master lease for master j held by master i expires, master i mark all tasks in S_j as unallocated. When a task completes, it is removed from S_i or R_i.

A task allocated from S_i by master i only causes redundant task executions when a worker lease expires. A task allocated from R_i by master i always cause a redundant task execution, unless a remote master j has crashed before allocating this task in S_j. A task $u \in S_j$ in R_i will only be allocated after the master lease of master j has expired.

At each task completion (and the corresponding task allocation) and at the frequency $pmax$, a master broadcasts its state. This state includes which tasks have been completed, which tasks have been allocated, the task results, and master lease information to the other masters. Broadcasting

at task completion reduces the chance of two masters both allocating the same tasks in their remote sets. Broadcasting at *pmax* frequency increase the chance of overcoming temporary communication failures. The value of *pmax* is a function of the average task execution time and the value for the master lease. For instance, if a task takes c seconds to complete on average and given that $D(G)$ is almost always 2 or less, then *pmax* should be set to $c/2$ to ensure that state about individual task completions is broadcast at least twice. However, for a master lease m, if $c/2 > m$, *pmax* should be set to a smaller value than $c/2$ to avoid a master leases expiring first.

Redundant task execution occurs in WAMW when leases expire. Compared to AX, the number of redundant task execution can be large. If communications are truly asynchronous, then in the worst case even with *no* partitions the number of tasks executed is $W_{wc} = NK$ where N is the number of tasks and K is the number of workers. Even if we assume that the leases are well-chosen—that is, a lease on master i held by master j expires if and only if i and j partition from each other—then the number of tasks executed can be as high as $W_{sc} = N(1 + M_1 \times M_2/M)$ when the M masters partition into two components, one of size M_1 and the other of size M_2. These values are derived in Appendix B. We expect, of course, that communications will not as poorly behaved as are needed to attain these bounds. Assuming that we can assign lease values in an appropriate manner, we expect that the number of redundantly executed tasks can be kept small.

5.3 Assigning Leases

The value for master leases influences both the number of redundant tasks executed and the total execution time of all tasks. Assume that all masters work in isolation during the whole computation, causing W_{wc} tasks to be executed. For the total execution time of the computation to be optimal, the master lease should not be larger than the time t it takes to execute the local set S_i for a master i, since the master immediately starts executing tasks in R_i once S_i is completed. Hence, t seems like a reasonable upper bound for the master lease. For long-lasting partitions, though, t is unrealistically large to use for master leases, since doing so will cause the total execution time to become very large. The question is then, how small can the master lease be to make the number of redundant tasks small while at the same time to avoid blocking the computation unnecessarily.

One obvious value for a master lease is the expected duration of a partition. In general, such

a value is impossible to establish, but we can use a value based on our traces. Partition durations in the RON trace were less than 2 minutes for nine out of 11 partitions. For the exp1 trace, one partition lasted five minutes and one for over an hour. Hence, if we disregard the very long partitions, partition in the traces typically endured for five minutes or less. We thus set the master lease to ten minutes for our experiments to mask the majority of partitions while ensuring that the computation does not block for long periods during long-lasting partitions.

The value for the worker lease is based on the expected execution time of a given task. We assume that for most cases when a worker sends a RENEW_LEASE message to the master, the message get to the master successfully and in a timely manner, since all communication between workers and masters is within LANs with small latency and low packet loss. Thus the accuracy of the worker lease is not important, as workers can periodically renew the lease if it is an underestimate. In our simulations we assume that workers do not crash, and hence the worker lease has no significance for our results.

5.4 Simulation

We used the RON traces to determine whether a message could be successfully sent from one host to another during an unreliable broadcast. If a message sent from node i to node j in the RON trace fails, then we assume that all subsequent messages sent in WAMW from node i to node j fail until a message is successfully sent from node i to node j in the RON trace.

Since the task execution times are identical, we added a small random jitter in the interval $[0, 2]$ to the execution time of the tasks to avoid unrealistic synchronous behavior in master to master communication. Adding jitter causes a master state broadcast for all task completions (and corresponding allocations). Since network jitter was not part of the simulation for AX, we normalized the execution time for WAMW to exclude the additional time caused by the jitter.

To increase tolerance to communication failures when doing the state broadcasts in the master, additional broadcasts are sent every $pmax$ seconds. In our experiments, tasks have lengths $100i$ seconds where $i = 1, \ldots, 5$ and to ensure we have at least one additional broadcast between every task completion broadcast we set $pmax$ to $100/2 = 50$. Note that we do not increase $pmax$ for $i > 1$ to avoid master leases from expiring due to temporal communication failures. We assume

		$Sim(1)$	$Sim(2)$	$Sim(3)$	$Sim(4)$	$Sim(5)$
Fastest running time		700	1,400	2,100	2,800	3,500
Start	Slowdown	1	1	1	1	1
at	Total msgs	18,360	21,720	25,080	28,440	31,800
500,000	Pct msgs failed	0.02%	0.02%	0.06%	0.08%	0.11%
Start	Slowdown	1	1	1	1	1
at	Total msgs	18,360	21,720	25,080	28,440	31,800
510,000	Pct msgs failed	0.08%	0.10%	0.19%	0.23%	0.28%
Start	Slowdown	1	1	1	1	1
at	Total msgs	18,360	21,720	25,080	28,440	31,800
520,000	Pct msgs failed	0.20%	0.16%	0.19%	0.21%	0.24%
Start	Slowdown	1	1	1	1	1.01
at	Total msgs	18,360	21,720	25,080	28,440	31,830
530,000	Pct msgs failed	7.40%	6.07%	6.38%	6.15%	6.02%
Start	Slowdown	1	1	1	1.02	1
at	Total msgs	18,360	21,720	25,080	28,485	31,800
540,000	Pct msgs failed	6.14%	6.05%	6.36%	6.03%	5.69%
Start	Slowdown	1.07	1.04	1.02	1.02	1
at	Total msgs	18,405	21,765	25,125	28,485	31,800
550,000	Pct msgs failed	11.82%	11.72%	11.79%	11.95%	11.82%

Table 7: WAMW simulation results from traces *with no* partitions.

that there is one master and ten workers on each of the 16 nodes monitored by RON, and that there are 1,000 tasks to be executed totally. The computation terminates when all masters know the results of all tasks. As with AX (see Table 2) we first ran the simulations over the same periods of time in the RON traces that we used for AX and reported in Table 2. The results are shown in Table 7.

We also report in the simulation results the number of messages that WAMW sends. The values are significantly larger than the number sent with AX. The average rate of message transmission is

	Partn 1	Partn 2	Partn 3	Partn 4	Partn 5
Start time	652,700	905,365	922,027	1,071,235	1,140,363
Duration	27,700	71	28	98	92
Execution time	10,000	700	700	700	700
Slowdown	14.3	1	1	1	1
Redundant tasks	1,000	0	0	0	0
Total msgs	78,480	18,360	18,360	18,360	18,360
Pct msgs failed	28.19%	3.56%	2.64%	4.85%	1.38%

Table 8: WAMW simulation results from traces *with* partitions.

faster for shorter tasks. With $Sim(1)$ the rate is about 26 messages/second and with $Sim(5)$ it is about 9 messages/second. If the messages are large, then the message overhead would most likely pose a scalability issue when task length is short.

The number of redundantly executed tasks is always zero, as expected given that there were no partitions. 80% of the slowdown factors are 1, and the maximum value is 1.07 which is much better than the performance of AX, where the *largest* slowdown was 11.17 and the *smallest* slowdown greater than 1 was 1.49. For the results where the slowdown is larger than 1, the execution time is always optimal plus $pmax$, where $pmax$ is 50 seconds. In these cases, the last task completion broadcast from a controller failed to reach all other controllers, but the successive broadcast which happens $pmax$ seconds later completed the broadcast.

We then ran $Sim(1)$ over the five different time intervals in Table 5. The results are shown in Table 8. Since the network is partitioned during the whole execution in the first simulation, all the tasks are executed redundantly, as they are with AX. For the other simulations, no tasks are executed redundantly, since the duration of the partitions are less than the master lease. The slowdown factor of the first simulation is 14.3, and that of all the other simulations are 1, all of them optimal.

To summarize, in most cases WAMW runs faster than AX, since it does not block during non-transitive communications. And in practice, the number of redundant tasks executed by WAMW is not greater than AX. Attaining this performance requires using good values for the lease times. In practice, a user might set the master lease times to balance off his or her own particular trade off between running time and redundant task execution.

6 Conclusions

We started this research because we wanted to know if we should use wide-area group communications in constructing a wide-area master-worker framework for GriPhyN. We chose a protocol that had been developed to minimize the amount of redundant tasks execute and evaluated it both with a simple analytical model and via simulation. The protocol was not meant to be a practical one, but on the surface it does not appear hard to make it more practical. We also compared it's performance under simulation against a simple protocol that does not use group communications

and that has a significantly higher upper bound on the amount of redundant work.

We were surprised by the frequency of nontransitive or asymmetric communications. In an earlier study [14] one of us had found significantly less periods of poor communications connectivity, but the network we considered contained only five nodes. From the traces we used to evaluate AX, it appears that as the number of nodes grow, the more likely it will be that one will encounter periods of nontransitive or asymmetric communications. Hence, using group communications systems upon which to build wide-area master-worker does not seem a good idea.

One can reduce the impact of poor communications connectivity in two ways. One way is to identify off-line which communication links will prove to be unreliable and then configure the backbone of the group communications system to avoid these links. An example of doing this is shown in [14]. The drawbacks of this approach are (1) it isn't clear how accurately and completely troublesome links can be identified in advance, and (2) the graph may have small node and link cut-sets, thereby increasing the chances of a partition. For example, in [14] to avoid a relatively unreliable link, we set up the backbone as a tree. The failure of one node subsequently caused a long-lasting partition.

The other way to reduce the impact of poor communications connectivity would be to relay information across better-performing links. This is what overlay networks like RON do [2] and one group membership protocol did [19]. Scalability of such overlay networks is not clear and is an ongoing research problem; if it can be addressed, then this would be a possible approach. However, our initial results with the simple algorithm indicate that the kind of poor connectivity that exists is masked quite well by limited flooding (and presumably by gossip protocols). Hence, it isn't clear that the generality of an overlay network is required. If one had other reasons for wanting partitionable group communications, then it would be worth considering using very limited gossip or flooding underneath.

What group communications offers for wide-area master-worker computation is a mechanism to ensure that in the worst case, the number of redundantly executed tasks is small. Our initial experience with a simple protocol, though, indicates that the expected number of redundantly executed tasks can be kept quite small without using group communications. If the worst-case scenario were more dire—for example, as it is in the Bancomat problem [20], then it would be worthwhile

running a group communications system over a communications layer that relayed information to avoid poor links. For wide-area master-worker, it is hard to justify such an expenditure of effort to avoid a highly unlikely worst-case behavior.

We believe, though, that group communications has a role in wide-area master-worker. One would wish to replicate masters to a small degree within each local area network. This would be done both to balance load and to mask crash failures of masters. Since communications is more reliable in a local area network, using a tightly-replicated state approach within a local-area network should work well.

Acknowledgments

We would like to thank David Anderson, Frans Kaashoek, Omar Bakr, and Idit Keidar for giving us access to their traces. We would also like to thank Omar Bakr, Idit Keidar, and Alex Shvartsman for fruitful discussions about the research reported here.

References

[1] Special Issue on Group Communication Systems. In *Communications of the ACM*, volume 39(4), April 1996.

[2] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Symposium on Operating Systems Principles*, pages 131–145, 2001.

[3] David G. Anderson. Personal communication, June 2002.

[4] O. Babaoglu, A. Bartoli, and G. Dini. Enriched View Synchrony: A Paradigm for Programming Dependable Applications in Partitionable Distributed Sytems. In *IEEE Transactions on Computers*, volume 46(6), pages 642–658, June 1997.

[5] Ozalp Babaoglu, Alberto Bartoli, and Gianluca Dini. Programming Partition-Aware Network Applications. In *Advances in Distributed Systems*, pages 182–212, 1999.

[6] Ozalp Babaoglu, Renzo Davoli, Alberto Montresor, and Roberto Segala. System Support for Partition-Aware Network Applications. In *International Conference on Distributed Computing Systems*, pages 184–191, 1998.

[7] Omar Bakr and Idit Keidar. Evaluating the Running Time of a Communication Round over the Internet. In *21st ACM Symposium on Principles of Distributed Computing*, pages 243–252, 2002.

[8] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning, 1996.

[9] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. In *ACM Transactions on Computer Systems*, volume 17, May 1999.

[10] F. Cristian and C. Fetzer. The Timed Asynchronous Distributed System Model. In *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, June 1999.

[11] C. Georgiou, A. Russell, and Alex A. Shvartsman. The Complexity of Distributed Cooperation in the Presence of Failures. In *Proceedings of the International Conference on Principles of Distributed Computing*, 2000.

[12] C. Georgiou and Alex A. Shvartsman. Cooperative Computing with Fragmentable and Mergeable Groups. In *7th International Colloquium on Structural Information and Communication Complexity*, pages 141–156, 2000.

[13] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[14] Idit Keidar, Jeremy B. Sussman, Keith Marzullo, and Danny Dolev. Moshe: A Group Membership Service for WANs. In *ACM Transactions on Computer Systems*, August 2002.

[15] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. In *Communications of the ACM 21*, volume 21, pages 558–565, July 1978.

[16] Meng-Jang Lin, Keith Marzullo, and Stefano Masini. Gossip Versus Deterministically Constrained Flooding on Small Networks. In *14th International Symposium on Distributed Computing*, 2000.

[17] Grzegorz Greg Malewicz, Alexander Russell, and Alex A. Shvartsman. Distributed Cooperation During the Absence of Communication. In *International Symposium on Distributed Computing*, pages 119–133, 2000.

[18] Grzegorz Greg Malewicz, Alexander Russell, and Alex A. Shvartsman. Optimal Scheduling for Disconnected Cooperation. In *Proceedings of VIII International Colloquium on Structural Information and Communication Complexity*, 2001.

[19] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant, distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, October 1995.

[20] Jeremy B. Sussman and Keith Marzullo. The Bancomat Problem: An Example of Resource Allocation in a Partitionable Asynchronous System. In *International Symposium on Distributed Computing*, pages 363–377, 1998.

A WAMW Pseudocode

```
def worker:
  result = NULL
  while true:
    task = request_work_from_master(result)
    if task == NO_MASTER:
      terminate()
    lease_timeout = extract_lease_timeout(task)
    fork(task)
    task_completed = false
    while not task_completed:
      result = block_until(expired(lease_timeout) or task_completed)
      if result == LEASE_EXPIRED:
        lease_timeout = renew_lease(task)
      else:
        task_completed = true
```

Pseudocode A.1: Worker main loop.

```
def request_work_from_master(result):
  controller = first available master for this worker
  while true:
    task = task_request(master, result)
    if task == NETWORK_FAILURE:
      mark_as_unavailable(master)
      master = next available master for this worker
      if not master:
        return NO_MASTER
    else:
      return task
```

Pseudocode A.2: Worker to master request handler.

```
def master:
  pmax_interval = interval between each master state update
  pmax = current_time() + pmax_interval
  idle_workers = []
  while true:
    state_update = false
    type, worker, result = get_worker_request()
    if type == RENEW_LEASE:
      set_worker_lease(worker)
    if type == TASK_REQUEST:
      idle_workers.append(worker)
      if result != NULL:
        state_update = true
        mark_task_as_done(result)
    while len(idle_workers) > 0 and num_unallocated_tasks() > 0:
      task = allocate_task()
      worker = idle_workers.pop()
      set_worker_lease(worker)
      send_task_to_worker(task, worker)
      state_update =true
    update_master_state(state_update)
    check_leases()
    if all_tasks_completed():
      terminate()
```

Pseudocode A.3: Master main loop.

```
def update_master_state(state_update):
  if current_time() > pmax or state_update:
    pmax = current_time() + pmax_interval
    send_local_state_to_masters()
  state = receive_state_from_masters()
  if state:
    update_local_state(state)
```

Pseudocode A.4: State updates from master to the other masters.

```
def check_leases:
  foreach master_lease:
    if expired(master_lease):
      unallocate_all_tasks_allocated_by_master()
  foreach worker_lease:
    if expired(worker_lease):
      unallocate_task_allocated_by_worker()
```

Pseudocode A.5: Checking for expired leases in the master.

B Worst-case Analysis of WAMW

Let:

M be the number of masters.

K_i be the number of workers associated with master i; $K_i > 0$.

K be the number of workers: $K = \sum_{i=1}^{M} K_i$.

N be the number of tasks.

W be the total number of tasks executed.

Since communication is asynchronous, we can delay messages for an arbitrary amount of time without having a partition (assuming that we define a "partition" to mean communications is broken for an even longer time, perhaps forever). In this model, W can be as large as NK: Delay all messages between masters so that the master leases expire. Thus, each master will compute all N tasks. Consider master i. Since it is work-conserving, it will assign K_i tasks in each round. Number the workers as w_1 through w_x where $x = K_i$. We can delay the replies long enough so that the worker leases expire. The master will not assign a task until a worker responds with results of a task; let this worker be w_1. Thus, the task that w_1 computed will not be redone. We can reassign all the other previously assigned tasks, for example, by assigning W_{sc}' task to w_1, w_3's tasks to W_{sc}, and so on.

The total number of tasks that master i computes is the number of tasks w_1 computes plus the number of tasks W_{sc} computes ... plus the number of tasks w_x computes, which gives a worst case of:

$$N + (N-1) + (N-2) + \ldots + (N - K_i + 1) = K_i(2N - K_i + 1)/2)$$

The total number of tasks W_{wc} is thus:

$$W_{wc} = \sum_{i=1}^{M} K_i(2N - K_i + 1)/2 = NK + K/2 - \sum_{i=1}^{M} K_i^2/2$$

From this it follows that W_{wc} is maximized when $\forall i : K_i = 1$ in which case $W_{wc} = NK$.

Suppose we adopt a stronger model: Communications can be arbitrarily delayed only if there is a partition, and partitions are detected by timeouts of master leases. For example, if a set of masters c_1, c_2, c_3, c_4 send messages to each other and then partition into two components c_1, c_2 and c_3, c_4, then by the time the partition is detected c_1 and c_2 have received each other's messages (but not necessarily c_3 and c_4's messages). We also assume that workers communicate with their masters in a timely manner and so no worker leases expire

Under this model, W still depends on more than the number of installed views. Consider the simple case of the M masters partitioning into two: One side, A, consists of M_1 masters and the other side, B, consists of M_2 masters. Assume that each master has the same number of workers. Have the masters each complete their slices, but delay all messages they send among themselves. After the partition, each master in A will redo the tasks that were computed in B and vice versa. So, before the partition the A side computed NM_1/M tasks and the B side computed NM_2/M tasks. Afterwards, the A side computes $M_1 \times NM_2/M$ tasks and the B side computes $M_2 \times NM_1/M$ tasks. Summing these, we get the total number of task computations as, and so the work is:

$$W_{sc} = N + 2NM_1M_2/M$$

W_{sc} is maximized when $M_1 = M_2 = M/2$, giving in the worst case $W_{sc} = (1 + M/2)N$. Even with only only two views installed, the worst-case of W_{sc} scales with the number of masters and is thus not optimal.