Parallel Processing Letters

SCHEDULING ISSUES IN SURVIVABILITY APPLICATIONS USING HYBRID FAULT MODELS*

AXEL W. KRINGS, WILLIAM S. HARRISON
Computer Science Department, University of Idaho
Moscow, Idaho 83844-1010, USA

and

AZAD AZADMANESH
Computer Science Department, University of Nebraska at Omaha
Omaha, Nebraska 68182-0500, USA
and

MILES McQUEEN
Idaho National Engineering and Environmental Laboratory (INEEL)
Idaho Falls, Idaho 83402, USA

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

This research addresses scheduling issues in networked computer systems with survivability requirements, i.e. systems in which essential services must survive malicious acts. In order to achieve survivability based on spatial redundancy, agreement algorithms are needed as a mechanism to consolidate results of individual replicas. The potentially enormous overhead associated with communication and voting schemes of the algorithms put unique burdens on the scheduler as the efficiency of scheduling determines the suitability of the agreement algorithm. This paper derives agreement task graphs, representing computations, and inter-process communication based on phantom tasks. Task graph primitives are identified and it is shown how their scheduling directly influences the performance of the agreement algorithm. Finally, the notion of dynamic *k-of-N* precedence is introduced and its impact on resource reclaiming for early stopping algorithms is discussed.

1. Introduction

With malicious computer attacks, i.e. hacking, reaching epidemic proportions, issues of computer and network survivability have surfaced in recent years. Of real

*This research was supported in part by an LDRD grant form the Idaho National Engineering and Environmental Laboratory (INEEL) and grant 60NANB1D0116 from the National Institute of Standards and Technology (NIST).

concern is the reliance of critical applications on networked computer systems. Failure or compromises of such systems could cause threats to national infrastructures or lead to catastrophe, e.g. loss of life, damage to the environment, or unacceptable financial losses. It is therefore of great importance to consider the resilience of system under attack.

Survivability is a new area of research concerning the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents [11]. Other definitions focus their attention to satisfy survivability of certain critical requirements, addressing specific requirements for security, reliability, real-time responsiveness, and correctness [25].

The key assumption in system survivability is that *anything* is possible. One therefore assumes that intrusions will occur sooner or later. When considering effects of intrusions, one has to recognize that in a computer any mechanism that empowers can (and will) be used against you, e.g. root or administrator privileges. This is different from fault-tolerance, which addresses dependability considering issues such as aging of components, system environment, or electrical interference. Assumptions are made about the statistical probabilities of certain events, e.g. it is assumed to be much more likely for a component to fail in a safe state than to fail in a specific asymmetric way causing pathological behavior. In survivability, this assumption does not hold. Attacks are assumed to be malicious rather than benign, and the probability of an attack does not necessarily follow predictable patterns. For example, if a vulnerability has been found in a specific system, but has not been published anywhere, the probability of this system being successfully attacked using the vulnerability is probably very small. However, posting the system's vulnerability in "hacker news groups" will almost certainly result in a successful attack.

Even though in survivability the statistical assumptions about faults are not the same, one should keep in mind the general principles of fault-tolerance under different fault models when considering methods to avoiding or surviving attacks and their effects. This argument holds if one views a system as a "black box".

Many different types of faults have been defined, some having orthogonal properties [1,18]. For example, fail-stop behavior implies that the faulty processor ceases operation and alerts other processors of this fault. Crash faults, on the other hand, assume that the system fails and looses all of its internal state, e.g. the processor is simply down. One speaks of omission faults when values are not delivered or sent, e.g. due to a communication problem. If outputs are produced in an untimely fashion, then one speaks of a timing fault. Transient faults imply temporary faults, e.g. glitches, with fault free behavior thereafter. If transient faults occur frequently, one speaks of intermittent faults. This set of fault types is by no means complete and serves only as a basic introduction. The definition of faults seem to change with the application domain. For instance, fault models suitable for computer dependability may not necessarily match the behavior of network and computer security applications [1].

Whereas the previous paragraph considers different types of classical faults, their behavior with respect to other processors can be described in simpler models which have been used with respect to replication and agreement algorithms. Specifically, fault models have been considered whose main behavior types are *benign*, i.e.

globally diagnosable, *symmetric*, i.e. faulty values are seen equal by all non-fault processes, and *asymmetric* or malicious, i.e. there are no assumptions on the fault behavior [21,27].

Even though much research has been conducted that considers issues of fault-tolerant scheduling, the types of faults addressed are actually small. The main focus has been on benign and transient faults [12,14,22,23]. With respect to fault-tolerance, *k-fault-secure scheduling* [4,15,16,17] has been introduced which implies that the system can tolerate up to k faults. However, again the research consider only permanent and transient fault models.

This paper addresses scheduling issues beyond the context of benign and transient faults to include the full spectrum of faults represented in hybrid fault models. The research focus is on survivability applications which have to deal with multiprocessor scheduling as part of redundancy management. The distributed workload of interest is defined by the agreement algorithm used to consolidate results of replicated tasks in the presence of malicious acts, i.e. malicious faults. Section 2 gives the motivation for spatial redundancy and introduces agreement algorithms and hybrid fault models. The scheduling and communication model are described in Section 3. For given agreement algorithms, agreement task graphs and task graph primitives are presented in Section 4. Section 5 focuses on performance issues. The scheduling of the graphs or primitives is shown to depends on the effective scheduling of certain groups of tasks. Finally, Section 6 concludes the paper.

2. Background

Survivability has been defined with respect to *Resistance*, *Recognition*, *Recovery*, and *Adaptation* [11]. The first three are referred to as "Three-R's". Resistance addresses intrusion prevention, Recognition implies intrusion detection, and Recovery addresses the need to return to a safe state after or during an attack or fault. Finally, Adaptation applies knowledge gained as a result of an attack in order to robustify the Three-R's.

2.1. Inherent Need for Redundancy

Survivability, as compared to security, views recovery as an integral part of the process. Recovery from loss of information, e.g. data, status, or context, implies redundancy. Redundancy, in turn, puts an extra burden on the scheduling environment with respect to redundancy management. The redundancy mechanisms available are "spatial", "information", and "time redundancy". With respect to the scheduler, spatial redundancy presents the greatest challenges, requiring multiprocessor scheduling and the associated problems of data movement and communication delays. The redundancy management indirectly implies the need for information exchange in order for redundant components to agree on correct values or actions. Time redundancy requires scheduling the same tasks multiple times on the same processor, however, no communication is required. Information redundancy only cause additional computational overhead to process the additional information, e.g. error correction codes. In the rest of this paper we refer to spatial redundancy only.

2.2. Agreement Algorithms

Redundancy is the assumed mechanisms to allow for recovery from the affects of faults and malicious acts. In order to detect or mask any discrepancies among the redundant copies, voting mechanism must be in place. Agreement algorithms are the common solution. The algorithms considered range from simple majority voting to full Byzantine Agreement [20]. A wide pallet of agreement solutions exist, differing in the constraints put on the fault model and the networking environment. The main issues are the fault model considered, e.g. benign or malicious, the network topology, e.g. fully or partially connected graphs, the networking protocol, e.g. point-to-point or multicast, and communication synchronization, e.g. synchronous or asynchronous.

One key issue is the time of the agreement, addressed by the notion of *immediate* and *eventual* Byzantine agreement [8]. Immediate agreement implies that all processors agree at the same time. In eventual agreement, all processors will eventually agree, but perhaps at different times.

The fault model and the agreement algorithms used in the replication scheme define the number of processors as well as the extent of the communication. Typically, as the number of processors increases, there is potential for dramatic increase in communication. This has direct implications on the scheduler and the scheduling environment.

2.3. Hybrid Fault Models

A fault model partitions the fault space. Together with the agreement algorithm, satisfying certain algorithm specific assumptions, it defines the total number of processors N needed and the overhead with respect to inter process communication. Not only does N have direct implications on the cost of the distributed system, which scales linear in N, but it also defines the communication overhead. For agreement using point-to-point protocols, algorithms can have run time and message complexity of up to $O(N^m)$, where m is the number of malicious faults [20].

In fault-tolerant system design, hybrid fault models have been used in order to achieve higher reliability or lower degree of replication. Figure 1 gives an overview of fault models.

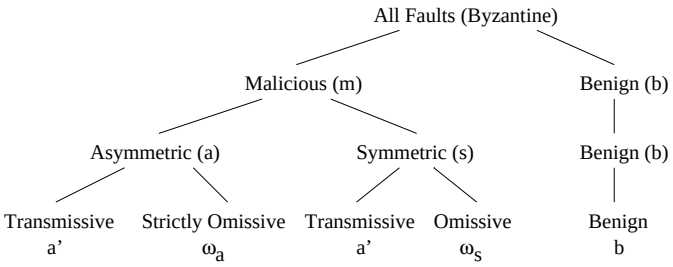


Fig. 1. Fault model

The simplest model is the single-mode Byzantine fault model at the root level. Every fault is assumed to be malicious, i.e. asymmetric, and the agreement algo-

rithms needs $N \geq 3t+1$ processors and $r \geq t+1$ rounds of communication, where t is the number of faults. An algorithm satisfying these requirements was introduced in [20].

At the second level, faults are partitioned into malicious and benign faults. This two-mode fault model was introduced in [24] and requires $N > 3m + b$ processors and $r > m$ rounds, where m and b are the number of malicious and benign faults respectively.

In the three-mode hybrid fault model at the next level malicious faults are furthermore partitioned into symmetric and asymmetric faults [21,27]. The model requires $N > 2a + 2s + b + r$ processors and $r \geq a$ rounds, where a and s indicate the number of asymmetric and symmetric faults respectively. In fault-tolerance this model thrives under the assumption that the fail rates λ for the individual fault types decrease drastically with their respective fault complexity, i.e. $\lambda_a << \lambda_s << \lambda_b$.

At the lowest level is the five-mode hybrid fault model of [2] which adds the notion of transmissive and omissive fault behavior. Transmissive asymmetric behavior implies that conflicting values are delivered, strictly omissive asymmetric implies that the correct value is received by at least one, but not all. Transmissive symmetric assumes the delivery of the same erroneous value, whereas omissive symmetric fails to deliver the value to any receiver. The result is $N > 3a'+2s'+3\omega_a+2\omega_s+b+1$.

3. Scheduling and Communication Model

The application workload we are interested in is the computation and communication associated with the agreement algorithms. Unless otherwise indicated, we will limit the scheduling considerations strictly to the agreement induced workload.

The application environment consists of Linux based computers in a general local area network (LAN). Thus, we assume a heterogeneous processor environment consisting of N processors or computers and a network topology that is logically fully connected. As such, it can be a partially or fully connected graph or a bus. The connectivity of the topology is assumed to be at least unity. Note that this guarantees that the network is connected, but it does not imply fault-tolerance, e.g. link or processor failures could result in partitioning the network. Otherwise higher connectivity must be assumed [3,20].

The smallest unit of computation considered is a task T, which may be preempted during execution. The application to be scheduled consists of a task graph $G = (V, E)$, where V is the set of vertices, i.e. tasks T_i, and E is the set of edges representing the precedence relation among tasks.

We consider a hybrid scheduling model which assumes two kinds of tasks. Task set V consists of *real* and so-called *phantom* tasks. In [7] real tasks are defined as tasks in the usual sense, whereas phantom tasks are special tasks which consume time, but unlike real tasks, they consume no resources. Phantom tasks will be our primary mechanism for modeling communication. Therefore, we adjusted the definition in [7] slightly to reflect that a phantom task takes time but no CPU resources. This is motivated by a view of a network interface card as an autonomous identity that is not CPU bound. Throughout the paper, real tasks will be depicted by circles and phantom tasks by rectangles.

In general, two types of communication are considered, synchronous and asyn-

chronous. Strictly speaking, in the synchronous model, communication primitives operate in perfect synchronization, i.e. in lock-step. In a real system, such assumption is unrealistic. An asynchronous communication scheme, on the other hand, makes no assumption about the timing of communication. It is therefore difficult or impossible to differentiate between a communication unit, i.e. a message, that is late and one that got lost. Realistic networks operate under asynchronous communication assumptions with bounded communication delays.

There are basically two communication primitives in typical LANs. The first is point-to-point and is typically represented by sockets, pairs of communicating ports in the Transport Control Protocol (TCP). The second primitive is multicast, which constitutes communication to a set or subset of computers in a network. Broadcast is a special case of multicast addressing all nodes on the network. The most common broadcast protocol is User Datagram Protocol (UDP). Our primary focus is on point-to-point communication.

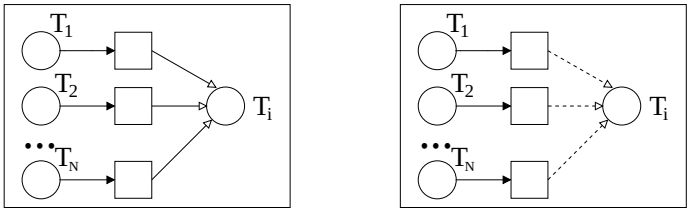
4. Agreement Task Graphs

In order to execute agreement algorithms, the associated tasks, represented by an agreement task graph, need to be scheduled in the multi-processor environment.

4.1. Benign Faults & Conditional Precedence

If only benign faults are considered, the resulting system can be represented as a parallel Reliability Block Diagram (RBD) with N blocks. This defines a *1-of-N* system, i.e. all N components must fail to cause a system failure.

The general model is the *k-of-N* model which requires at least k of the N modules to function fault free. Such model is very common in reliability analysis, and is referred to as a *k-of-N* system. The task graph of an *N-of-N* and *k-of-N* system are shown in Figure 2. Recall that squares denote phantom tasks modeling communication. Even though both graphs are the same, i.e. they have the same vertex and edge set, the graphs present different scenarios with respect to the release of task T_i. For the *N-of-N* graph, task T_i is released at the time the last of the predecessors finishes. This is different for the *k-of-N* graph, in which T_i is released at the time the first k predecessor tasks finish. Such behavior is referred to as *conditional precedence* [13], and is depicted in the graph by dashed edges. The concept of dynamic conditional precedence will be described in Subsection 5.3.



a) N-of-N b) K-of-N

Fig. 2. Conditional precedence

4.2. Malicious Faults

Malicious faults for point-to-point communication environments have been addressed in [20]. An Oral Message algorithm $OM(t)$ was presented that could deal with t faulty processors, assuming a total number of processors of $N \geq 3t+1$. The algorithm is an immediate algorithm and works in $t+1$ rounds of message exchange. Each node keeps track of values received in a so-called EIG-Tree [5], where each level represents a round of message exchange. Figure 3 shows the EIG-Tree of processor P_2 for $OM(2)$. Each node maintains such a tree. Edges represent messages received, leaf vertices contain the received value. Edge labels indicate from which processor a value has been received. For example, following the path from the root to the right most leaf node indicates that this leaf node received a value that originated at processor P_0, was sent to P_6 and lastly received from P_5 with value 0. The depicted EIG-Tree has one single leaf node at level 2, all other leaf nodes are at level 3. This stems from the fact that the tree is stored at processor P_2, which does not need confirmation about its own value from others.

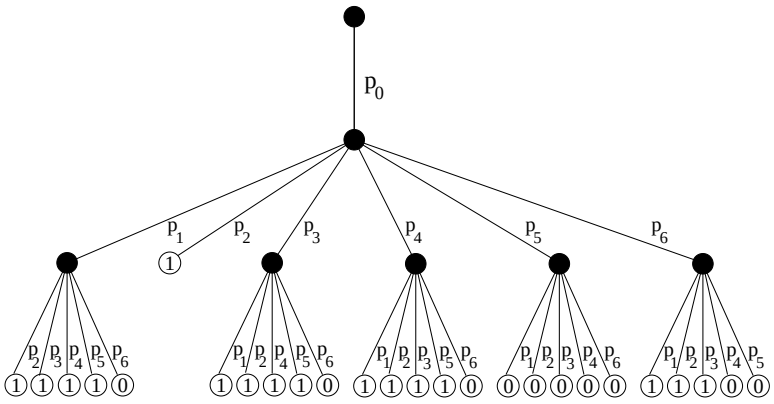


Fig. 3. EIG-Tree of OM(2)

The EIG-Tree embodies the computation and communication tasks that need to be scheduled in order to execute an agreement algorithm. Therefore, EIG-Trees build the basis for the agreement task graphs to be introduced below. Whereas computations involve updating of data structures, e.g. EIG-Tree entries, the communication is under the control of the network interface card. Therefore communication is assumed not to be CPU bound, which justifies its representation as a phantom task.

4.2.1. Task Graphs for OM(1)

In order to generate the task graph for an agreement algorithm let us consider an example task graphs for $OM(1)$. Figure 4 shows the task graph for round 0. On the left side, P_0 sends out messages, indicated by phantom tasks, to all other processors. This graph is extended on the right side, reflecting that the network interface of the other processors receive the messages and passes them on to the processor. The task graph after round 1 is captured in Figure 5. Note that each phantom task is

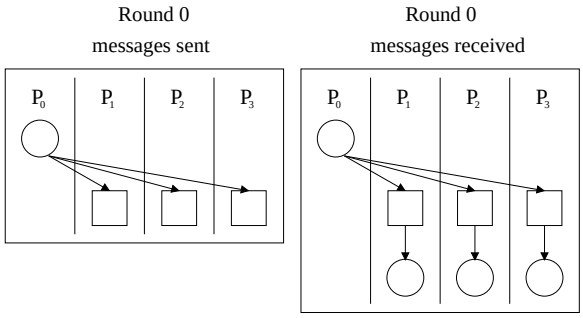


Fig. 4. Task graph: $OM(1)$ round 0

incident to exactly one edge, i.e. each phantom task has one incoming edge. To avoid visual clutter, multiple phantom tasks are depicted as arrays of rectangles, each element representing one phantom task.

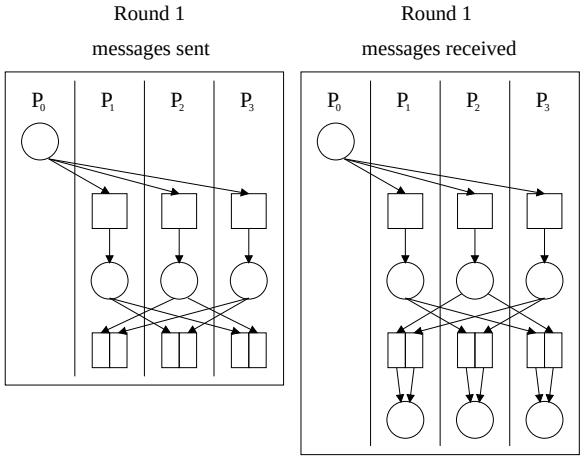


Fig. 5. Task graph: $OM(1)$ round 1

The extent of the number of messages passed becomes more apparent as the number of faults increase, as will be shown by going from $OM(1)$ to $OM(2)$, which requires $r = t + 1 = 3$ rounds of communication. The resulting task graphs after each round are shown in Figure 6 and Figure 7. It should be noted that in the last round of communication in Figure 7 each edge represents $N - r$ messages. If one needs to consider each message separately, then each edge and the corresponding phantom task has to be replicated $N - r$ times. This is in consequence to the algorithm, which always sends each message received to all other processors that have not received it yet.

4.2.2. Simplex versus Multiplex

The previous discussions showed task graphs for scenarios in which one single

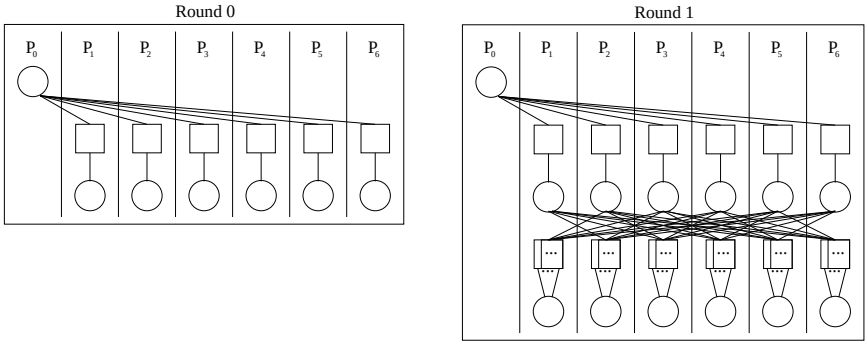


Fig. 6. Task graph: $OM(2)$ round 0 and 1

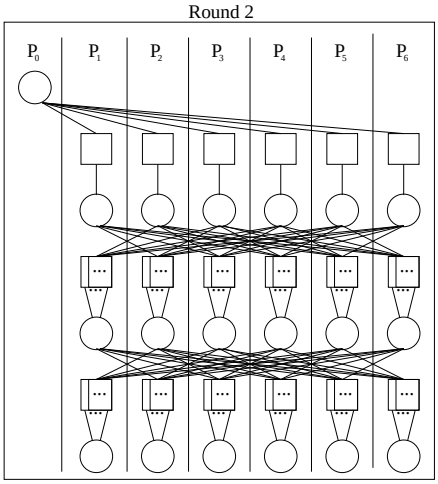


Fig. 7. Task graph: $OM(2)$ round 2

processor, i.e. P_0, distributed a value which every processor had to agree on. Such scenario is called *simplex*. This of course does not constitute fault-tolerance, since validity of the value cannot be guaranteed, e.g. if the originating processor lies to the other processors it can manipulate which value all good processors should agree on.

To satisfy fault-tolerance, it is assumed that each node in the redundancy configuration executes an agreement algorithm, i.e. each processor executes a simplex agreement. The final result is that each non-faulty processor has the same agreed upon values, one per simplex agreement. This *multiplex* scenario N-folds the number of messages sent, as each task at the top layer becomes the initiator of an agreement. Figure 8 shows the agreement task graphs for the simplex and multiplex scenarios of $OM(2)$.

4.3. Task Primitives

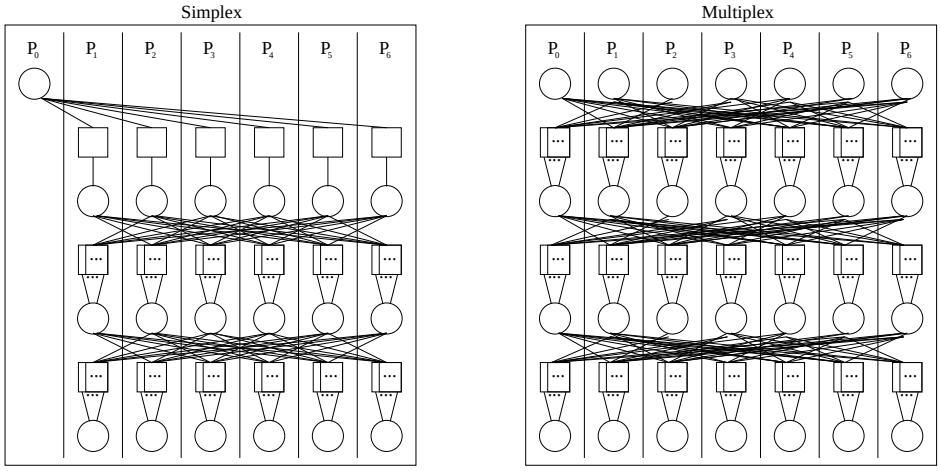


Fig. 8. Task graph: $OM(2)$ simplex and multiplex

All task graphs shown so far can be derived from three task graph primitives. The first primitive is *broadcast*, in which a real task sends out messages using phantom tasks. The broadcast primitive should not be confused with a broadcast or multicast protocol. Here the term broadcast only refers to the distribution, not the protocol. The second primitive is *receive*, in which a real task receives the messages from phantom tasks. The third primitive is *compute*, which manipulates the data structures, e.g. EIG-tree, and voting. The three task primitives are depicted in Figure 9. A forth primitive can be defined representing *k-of-N* precedence. This primitive looks just like the receive primitive, however, at run-time, it behaves different, as explained in Subsection 4.1. Scheduling these primitives effectively will increase performance of the algorithms, as will be shown in the next section.

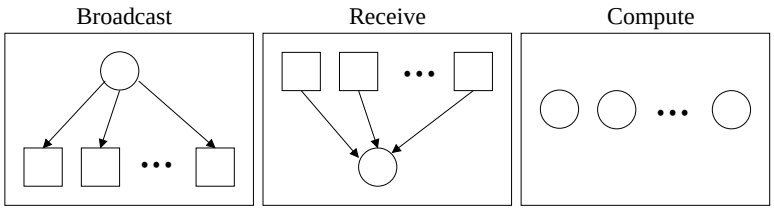


Fig. 9. Task graph primitives

5. Scheduling and Performance

The performance of an agreement algorithm under a specific environment can be measured by the time needed for reaching an agreement or by the communication overhead, e.g. the total number of messages sent. Performance is mainly affected by two factors: the processing power of each processor and the network speed.

The number of messages sent depends on the agreement protocol and the mes-

sage forwarding strategy [28]. For instance, a simple majority requires only a single round of communication, whereas on the other extreme the $OM(t)$ algorithm of [20] requires $t+1$ rounds of message exchange. Note that we are not considering multicast or broadcast operations since the focus of our research is mainly on networks using TCP/IP protocols.

5.1. Timing Issues

Given the non-deterministic behavior of most distributed environments, we focused our performance analysis of agreement on direct measurement. Even though the measurement of time is a very simple idea, the actual process is very complicated. The reason for this is twofold. First, in a distributed environment each processor has a different view of time. This is mainly due to differences in clock initialization times and differences in actual clock speeds, i.e. clocks usually drift apart slowly. With non-deterministic network delays, it is difficult to synchronize time. The second issue is the transparent behavior of operating systems in a multitasking environment. The problems associated with distributed clocks have been addressed extensively in the literature [6,10,26].

Figure 10 illustrates the different times of interest during the process of reaching agreement. Assume that T_{init} is the initiating task of a simplex agreement. At time t^s_{init}, the initiating processor begins to send out messages to all other processors in order to start an agreement. At times $t^s_{p1}, t^s_{p2}, ..., t^s_{pn}$ the replicated tasks $T_{p1}, T_{p2}, ..., T_{pn}$, executing on different processors, receive the value from the initiating processor respectively. Each processor starts its message exchange process and tries to reach an agreement. At times $t^e_{p1}, t^e_{p2}, ..., t^e_{pn}$, the processors make a decision and reach agreement. Each processor then sends a signal back to the initiating processor to inform it that an agreement has been reached. The initiating processor receives the last message at t^e_{init}. From the initiating processor point of view, the whole agreement took Δt_{init} time units. It is also possible that each processor will act immediately after it makes its decision, i.e. it does not need to report back to the initiating processor. Under this situation, the initiating processor does not know when an agreement is reached.

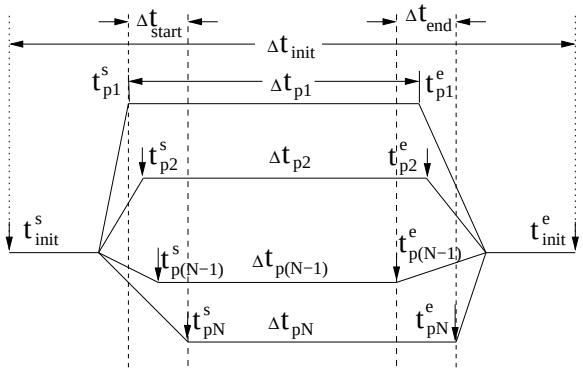


Fig. 10. Communication timing

If Figure 10 represents a majority voting scheme, then $t^s_{pi} - t^s_{init}$ is the communication time observed by processor P_i in the first (and only) round of communication. The communication phantom task T_i in the agreement task graph has thus an execution time equal to $t^s_{pi} - t^s_{init}$.

5.2. Scheduling for Performance

Of special interest with respect to performance are the intervals Δt_{start} and Δt_{end} between the dashed lines shown in Figure 10. Not all processors start and reach agreement at the same time. Interval Δt_{start} represents the maximum time difference among receiving processors when they become aware of the start of an agreement. If we assume that processors act immediately after they reach an agreement, then Δt_{end} indicates the maximum time difference among processors in taking action. This difference can be very important for many applications, since for Δt_{end} different processors can be in non-agreement states. If, for example, the agreement is in the context of database entries, then the distributed database is in an inconsistent state during this time.

Assuming a homogeneous processor environment, the single most important criteria affecting performance of agreement algorithms is the relative position of related real tasks across processors. For the $OM(2)$ algorithm, Figure 11 identifies these groups of tasks in the shaded ellipses. If one wants to minimize Δt_{end}, the leaf tasks have to be considered as a group as well.

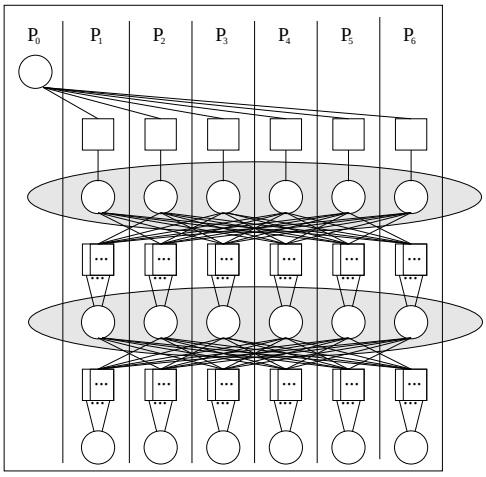


Fig. 11. Task groups in $OM(2)$

In general, task groups can be derived directly from the task graph primitives, as shown in Figure 12. The effectiveness of the agreement algorithm depends on how closely aligned these tasks can be scheduled across the processors. Whereas the groups in the broadcast and receive primitives are phantom tasks, the compute group consists of real tasks. In order to minimize communication, the phantom groups must be aligned. In the best case, this can be achieved by using a multicast protocol for agreement [3]. For point-to-point protocols one can force communica-

tion to all processors be an atomic operation within the operating system.

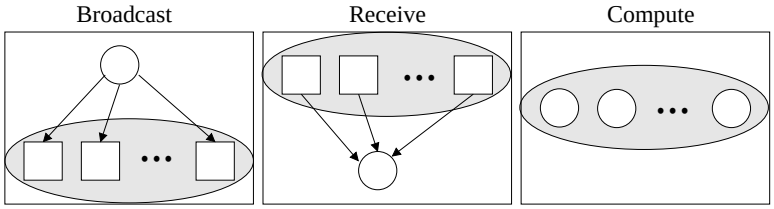


Fig. 12. Task groups of graph primitives

5.3. Dynamic k-of-N *Precedence*

In the simple *k-of-N* system of Subsection 4.1 k is a constant. In agreement algorithms with early stopping behavior [5,9,19], k is not known a priori but run-time dependent. These algorithms differ from the standard algorithms in that each processor agrees to a correct value as soon as a certain threshold of correct values is received. Early stopping algorithms are therefore eventual rather than immediate algorithms. Since the threshold determination is value driven, and therefore run time dependent, we define the notion of *dynamic k-of-N precedence* as a generalization of *k-of-N* precedence. Dynamic *k-of-N* precedence has big implications on processor and communication link reclaiming, e.g. slack-time reclaiming. The reason is that, whereas the entire task graph needs to be scheduled, upon the conditional release of a task T_i by k predecessors, the remaining $N-k$ predecessors can be aborted, thereby freeing up respective processor resources. Furthermore, aborting tasks reduces message complexity since these tasks refrain from sending messages. The impact of early stopping on the message complexity has been quantified in [28].

5.4. Experimental Results

Scheduling the groups of the task graph primitives conforms to the general notion that synchronizing the groups across processors gives the highest performance. This follows directly from the definition of Δt_{start} and Δt_{end} of Figure 10.

In an attempt to capture the real timing associated with the task groups indicated in Figure 12, an experiment was conducted that involved the execution of three different agreement algorithms in computer clusters of variable size. The number of computers in the cluster is equal to the size of the task group in the task graph primitives. Only in *k-of-N* precedence primitives might the actual number of tasks executed in the group be less than the cluster size, i.e. only if $k < N$.

The clusters consisted of 80X86 compatible computers running RedHat Linux version 6.0 and 6.2 with a standard TCP/IP protocol stack, configured using 10Base-T and a Cisco System Catalyst 1900 switch. In order to overcome the time measurement problems indicated in Subsection 5.1 the *xntp3-5.93* time synchronization package was utilized.

The agreement algorithms implemented were the classic $OM(m)$ algorithm by Lamport et. al. [20] and two derivations of early stopping agreement algorithms [19]. Whereas the Lamport algorithm utilized the task graph primitives of Figure 9, the

early stopping algorithms imply primitives with *k-of-N* precedence, where k is a threshold value depending on the data received at run-time. Two early stopping algorithms, *SyncES* and *AsyncES* were considered. *SyncES* implements early stopping with synchronized voting after every message in a round is present, whereas *AsyncES* is an asynchronous version where a voting process is started after each received message. Readers interested in the early stopping algorithms are referred to [19] and [28] for details.

Next, the cluster sizes were fixed. Given the number of processors required to reach agreement [20], i.e. $N \geq 3t + 1$, we considered the number of malicious fault to be $t = 1$, 2, and 3, resulting in cluster sizes of $N = 4$, 7, and 10, respectively.

Measuring the time intervals Δt_{start} and Δt_{end} in Figure 10 reveals how close the groups in the task primitives could be scheduled. The data presented next is based on repeating the executions of each algorithm 200 times. Figure 13 shows the maximum starting time differences Δt_{start} observed. The three algorithms show similar timing behavior with only slight differences among algorithms within the same cluster. However, with increasing cluster size the differences grow. It is expected that the overall Δt_{start} must increases with the cluster size as more tasks in the task groups must obtain their messages.

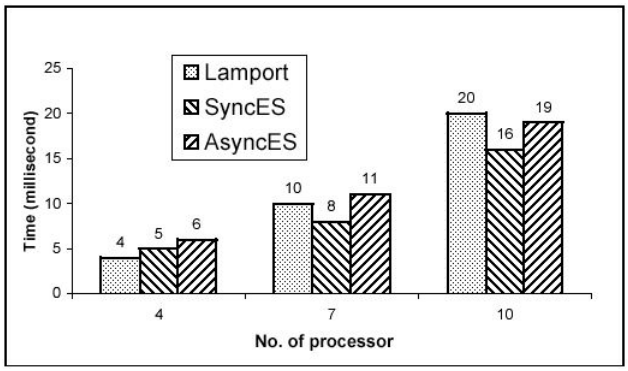


Fig. 13. Maximum starting time difference Δt_{start}

As each agreement executes, the impacts of the differences among the algorithms compound, resulting in looser scheduling of the task groups within the primitives. Figure 14 shows differences in Δt_{end} as the cluster grows. The small task graph primitives associated with the 4 processor cluster are tightly synchronized and show little differences in Δt_{end}. However, as the cluster grows the hight of the task graph, the size of the task groups in the primitives, and the number of messages increase. The time differences compound from each stage of the task graph to the next. The affect is a larger diversion in the ending time differences, which is most visible in the 10 processor scenario. Actually, the results are not surprising when one looks at the growth in the graph hight, which is linear to $t + 1$, and the task primitive group size, which is bound by $3t + 1$. Whereas the task graph for a 10 processor scenario is not shown due to its size, the growth for the 4 and 7 processor scenario can be seen by comparing Figure 5 and Figure 7.

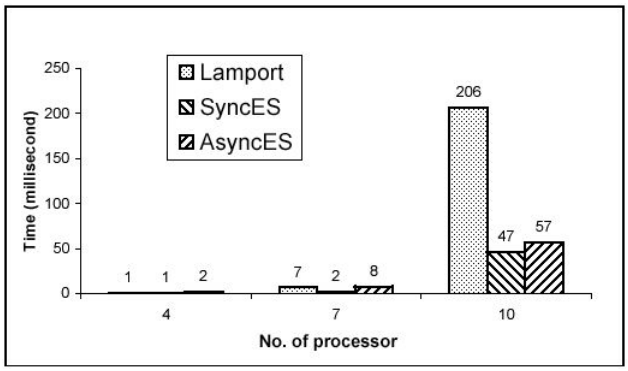


Fig. 14. Maximum ending time difference Δt_{end}

The significantly better Δt_{end} for the early stopping algorithms in larger clusters shows the strength of *k-of-N* scheduling as compared to Lamport's *N-of-N* primitives. It should be noted, that in a pathological scenario, the early stopping algorithms degenerate into Lamport's algorithm, i.e. in the worst case scenario $k = N$. The dynamic determination of k at run-time [19] increases resource reclaiming as k decreases since the size of the task groups within the primitives is reduced.

Lastly, the number of messages have to be considered when interpreting Δt_{end}. Figure 15 shows the total number of messages sent. In the 4 processor cluster the number is constant by the nature of the task graph and its execution. The power of *k-of-N* scheduling in the early stopping algorithms shows better performance as the cluster size increases.

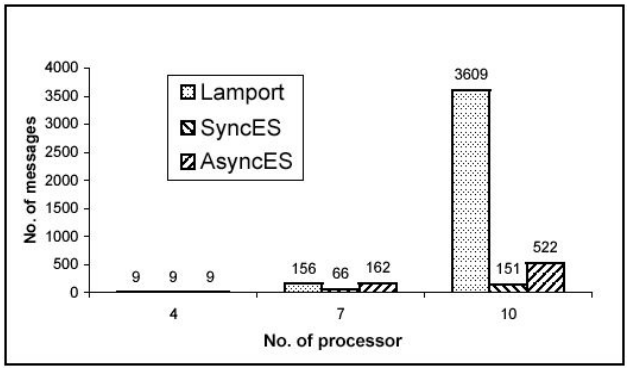


Fig. 15. Total number of messages sent

5.5. Fault Models and Task Graphs

After the establishment of agreement task graphs, their task graph primitives, and the performance implications based on scheduling the task groups, the impact

of the fault models needs to be addressed.

The most significant realization is that complex agreement algorithms are needed only if malicious faults can not be prevented. As a result, agreement involving only benign and symmetric faults can be achieved using majority voting. The simplex task graph of a majority voting algorithm involves a single round of communication. An example of such graph is the round 0 graph of Figure 6. If asymmetric faults are unavoidable, then complicated agreement algorithms must be employed. In the worst case, this results in task graphs similar to the one shown in Figure 11, which considered a 7 processor scenario under $OM(2)$.

The real strength of using the hybrid fault models shown in Figure 1 is that one can reduce the overall complexity or size of the agreement task graph by committing to a mixture of faults. For instance, in the three fault model addressed in [21,27], the number of rounds can be fixed to the number of asymmetric faults, independent of how many benign or symmetric faults should be considered. This has direct implications on the agreement task graph, whose depth is directly linked to the number of rounds. Addressing survivability, the fault mix can constitute a separation of (1) malicious faults caused by hackers and (2) the faults introduced by traditional dependability considerations. Whereas in the latter case asymmetric faults can be avoided by design [29], the hacker impact should not be assumed avoidable or preventable.

6. Conclusions

This research addressed scheduling of agreement algorithms used in survivability applications where recovery is based on spatial redundancy. For specific agreement algorithms, agreement task graphs, consisting of real and phantom tasks, were presented. Whereas real tasks represent CPU bound operations, phantom tasks model communication under the control of the network interface card. The effective scheduling of the agreement tasks graphs directly affects the performance and thereby feasibility of specific agreement algorithms. By using hybrid fault models, a mix of faults can be specified in order to reduces the complexity of the agreement and therefore the size of the associated agreement task graph. Task graph primitives for broadcast, receive and compute operations were identified. Within these primitives task groups were identified whose efficient scheduling is directly linked to the performance of the agreement. For early stopping agreement algorithms the agreement task graphs include dynamic *k-of-N* precedence, which allows for processor reclaiming.

References

[1] A. Avizienis, J.C. Laprie and B. Randell, Fundamental Concepts of Dependability, *Information Survivability Workshop (ISW-2000)*, Boston, Massachusetts, Oct. 24-26, 2000.

[2] M. H. Azadmanesh, and R. M. Kieckhafer, Exploiting Omissive Faults in Synchronous Approximate Agreement, *IEEE Transactions on Computers*, Vol. 49, No. 10, Oct. 2000, 1030-1042.

[3] O. Babaoglu, and R. Drummond, Streets of Byzantium: Network Architectures for Fast

Reliable Broadcasts, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, June 1985, 546-554.

[4] P. Banerjee and J. A. Abraham, Fault-Secure Algorithms for Multiple Processor Systems, *Proc. 11th International Symp. on Computer Architecture*, Ann Arbor, MI, June 1984, 279-287.

[5] P. Berman, J. A. Garay, and K. J. Perry, Optimal Early Stopping in Distributed Consensus, *Proc. 6th International Workshop on Distributed Algorithms (WDAG '92)*, LNCS 647, Springer-Verlag, Nov. 1992, 221-237.

[6] F. Cristian, and C. Fetzer, Fault-Tolerant Internal Clock Synchronization, *Proc. Thirteenth Symposium on Reliable Distributed Systems*, Dana Point, California, Oct. 1994, 22-31.

[7] J. S. Deogun, R. M. Kieckhafer, and A. W. Krings, Stability and Performance of List Scheduling With External Process Delays, *Real-Time Systems*, Vol. 15, No. 1, July 1998, 5-39.

[8] D. Dolev, R. Reischuk, and H. R. Strong, Eventual Is Earlier than Immediate, *23rd Annual Symposium on Foundations of Computer Science*, Chicago, Nov. 1982, 196-203.

[9] D. Dolev, R. Reischuk, and H. R. Strong, Early Stopping in Byzantine Agreement, *Journal of the ACM*, Vol. 37, No. 4, Mar. 1990, 720-741.

[10] S. Dolev, Possible and Impossible Self-Stabilizing Digital Clock Synchronization in General Graphs, *Journal of Real-Time Systems*, Vol. 12, No. 1, 1997, 95-107.

[11] B. Ellison, et.al., Survivable Network Systems: An Emerging Discipline, Carnegie Mellon University, Software Engineering Institute, Technical Report CMU/SEI-97-TR-013, 1997.

[12] S. Ghosh, R. Melhem, and D. Mosse, Fault-Tolerance Through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems, *IEEE Trans. Parallel and Distributed Systems*, Vol. 8, No.3, Mar. 1997, 272-284.

[13] D. W. Gillies, Algorithms To Schedule Tasks With And/Or Precedence Constraints, Ph. D. Thesis, University of Illinois at Urbana-Champaign, 1993.

[14] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel, Generation of Fault-Tolerant Static Scheduling for Real-Time Distributed Embedded Systems with Multi-Point Links, *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, San Francisco, USA, April, 2001.

[15] D. Gu, D. J. Rosenkrantz and S. S. Ravi, Construction and Analysis of Fault-Secure Multiprocessor Schedules, *Proc. 21st Intl. Symp. on Fault Tolerant Computing (FTCS-21)*, Montreal, June 1991, 120-127.

[16] D. Gu, D. J. Rosenkrantz and S. S. Ravi, Fault/Error Models and Their Impact on Reliable Multiprocessor Schedules, *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1992, 176-184.

[17] K. Hashimoto, T. Tsuchiya, and T. Kikuno, Fault-Secure Scheduling of Arbitrary Task Graphs to Multiprocessor Systems, *Proc. International Conference on Dependable Systems and Networks (DSN 2000)*, June 2000, 203-212.

[18] W. B. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems* (Addison-Wesley Publishing Company, New York, 1985).

[19] A. W. Krings, and T. Feyer, The Byzantine Agreement Problem: Optimal Early Stopping, *Proc. 32nd Hawaii International Conference on System Sciences*, No. stdds03, January 5-8, 1999, 1-12.

[20] L. Lamport, M. Pease, R. Shostak, The Byzantine Generals Problem, *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982, 382-401.

[21] P. Lincoln, and J. Rushby, The Formal Verification of an Algorithm for Interactive Consistency under a Hybrid Fault Model, *Computer Aided Verification, CAV'93*, Lecture

Notes in Computer Science, No. 697, Springer Verlag, 1993, 292-304.

[22] L. V. Mancini, Modular Redundancy in a Message Passing Systems, *IEEE Trans. Software Eng.*, Vol. 12, No. 1, Jan. 1986, 79–86.

[23] G. Manimaran and C. Siva Ram Murthy, A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis, *IEEE Trans. Parallel and Distributed Systems*, Vol. 9, No. 11, Nov. 1998, 1137–1152.

[24] F. J. Meyer, and D. K. Pradhan, "Consensus with Dual Failure Modes", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 2, April, 1991, 214-222.

[25] P. G. Neumann, Practical Architectures for Survivable Systems and Networks, (Phase-Two Final Report), Computer Science Laboratory, SRI International, June 2000.

[26] F. Schneider, Understanding Protocols for Byzantine Clock Synchronization, Department of Computer Science, Cornell University, Report 87-859, August 1987.

[27] P. Thambidurai, and Y.-K. Park, Interactive Consistency with Multiple Failure Modes, *Proc. 7th Symp. on Reliable Distributed Systems*, Columbus, OH, Oct. 1988, 93-100.

[28] G. Xiao, Early Stopping Byzantine Agreement in LAN based Network Environments, M. S. Thesis, Computer Science Department, University of Idaho, 2000.

[29] Y.C. Yeh, Triple-Triple Redundant 777 Primary Flight Computer, *IEEE Aerospace Applications Conference*, 1996, 293-307.