UNICON – A UNIversal CONsensus for Responsive Computer Systems*

Michael Barborak and Miroslaw Malek
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712

December 22, 1992

Abstract

The consensus problem is omnipresent and fundamental in multicomputer systems. Synchronization, communication, diagnosis, scheduling, reconfiguration and termination of a computation can all be solved by consensus. We propose a single, universal consensus protocol that is capable of solving all of these consensus problems in a single pass through a set of computers. The protocol's capabilities may be tailored to fault models ranging from those used in system diagnosis to the Byzantine Generals Problem. We believe that the UNIversal CONsensus protocol (UNICON) will become a foundation for responsive (fault-tolerant, real-time) multicomputer systems.

1 Introduction

Consensus is a condition of agreement. In particular, for fault-tolerant, distributed systems, the ability to reach consensus is the ability to share information among the population of fault-free processors despite the actions of the faulty processors. The information to be shared is irrelevant to a protocol that guarantees consensus as any data is ultimately a vector of binary digits. Therefore, protocols that result in a specific agreement, such as on a current time, should be suspected as being special cases of a general consensus algorithm. In other words, procedures to reach synchronization, reliable communications, diagnosis, membership, sensor stabilization, resource management, scheduling, replication management, reconfiguration, detection of a global state such as termination or any agreement on some aspect of the system status are restricted forms of a general consensus protocol. Thus we are motivated to identify a single algorithm with the combined functionality of each of these specific instances of consensus.

*This work was supported in part by ONR Contract N00014-88-K-0543, ONR Grant N00014-91-J-1858, IBM Agreement 203 and Texas Advanced Technology Program Grant 386.

In this paper we propose a UNIversal CONsensus (UNICON) protocol that possesses the flexibility to solve all of these specific instances of consensus. UNICON is a general mechanism for agreeing on data in a distributed environment. The data used in the agreement process is a function of what purpose a particular instance of UNICON is to serve. In other words, if UNICON is being used to synchronize clocks, then it is being used to agree on the values of local clocks. Therefore, the data are local clock values. Similarly, for reconfiguration, the data are the identifications numbers (ids) of fault-free processors. Alternately, the data could be a combination of values allowing synchronization, diagnosis, reconfiguration and more all in a single pass.

The use of UNICON in fault-tolerant computer systems is fundamental as UNICON eliminates any dependency on a single resource in a multicomputer environment.

For UNICON to perform consensus at every level of the operating system and user applications, it must work for a variety of fault models. At very low levels it might be necessary to assume Byzantine faults while at very high levels it might be sufficient to assume fail-stop faults [Schneider 1984]. Therefore, UNICON allows for the fault models of the processors involved in a particular instance of consensus to be specified. It can also handle the case of mixed fault models in a single consensus. This ability to adapt to the fault models in the system allows UNICON to remain efficient despite its wide application.

Using UNICON for support of a responsive (fault-tolerant, real-time) computer systems [Malek 1991] adds more requirements to the consensus task, namely, bounds on the execution time. It is necessary to know how long an instance of UNICON will need to complete, as well as the nature of the incomplete consensus results caused by a preemption of the UNICON task. Not only is this necessary for timely operation, but also for certain instances of consensus. For example, when synchronizing, a local clock value is highly dependent on how long ago it was sampled, and when diagnosing, there is a time dependent probability that the result is no longer correct, i.e., that some processor has since failed. Therefore, time has an integral role in UNICON.

In the remainder of this paper we shall explore the motivation, specification, implementation and application of UNICON. In Section 2 we look at the role of UNICON in distributed computations. In Section 3 we present the format for specifying an instance of consensus. In Section 4 we present an implementation of UNICON along with a consensus protocol designed to work in an environment of heterogeneous fault models. In Section 5 we look at the time analysis of UNICON as well as its application to clock synchronization and reliable communication. Finally, in Section 6 we give our conclusions.

2 A Model for Distributed Computation

Consider a distributed system comprising a number of processing elements (PEs) with an ability to send messages between themselves. A distributed computation is simply a group of these PEs working towards a common goal. We model a distributed computation as a series of two alternating operations, namely local execution and consensus. Much as Valiant's Bulk Synchronous Parallel model (BSP) considers a parallel computation to be a

series of alternating executions and synchronizations [Valiant 1990], a distributed computation involves each PE performing its portion of the computation (including recovery procedures) until it must send or receive information from the other PEs involved. Whatever is the nature of this information, an agreement is needed, i.e., consensus is executed. In other words, any communication is or is some part of an instance of consensus. UNICON is a recognition of this, as any non-local requirements of a PE are met by a particular instance of it. Therefore, we may model a distributed computation as a series of alternating local executions and calls to the UNICON process.

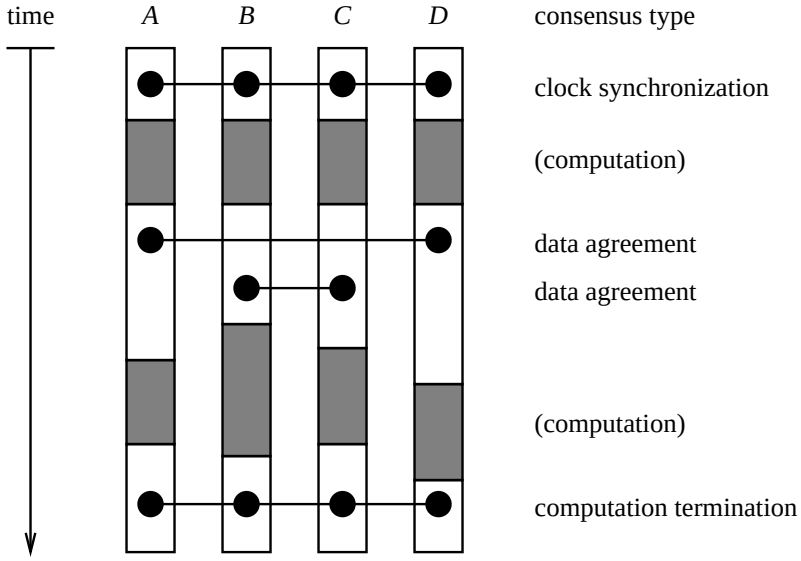

Figure 1: A computation/consensus view of a distributed program.

As an illustration, consider Fig. 1 which shows the execution profiles of four processors A, B, C and D. The column below a processor label represents when consensus is being performed (marked in white) and when a local computation is being performed (marked in gray). A horizontal bar is used to collect related executions of consensus. For example, the first consensus resulting in clock synchronization involves all four processors as marked by the horizontal bar connecting all four columns. As Valiant argues that the BSP model eases the programmer's task in writing parallel computations, we feel this computation/consensus paradigm will ease the development of responsive operating systems and applications.

A consensus-based approach towards the development of responsive operating systems was presented in [Malek 1992]. A primary rational is that we can never rely on a single resource thus it is necessary to progress via consensus. Therefore, the kernel of a responsive operating system consists of a number of consensus tasks as illustrated in Fig. 2 [Malek 1992].

By implementing this system with UNICON, we can obtain some definite advantages. First, we can encompass many differing fault models in the system model, and second, the operating system is based on a single algorithm. The result is a simplicity and functionality in the kernel that allows for more reliable coding, low installation overhead, and an ease of software maintenance. Moreover, we foresee an ease in estimating the timeliness of the

operating system tasks.

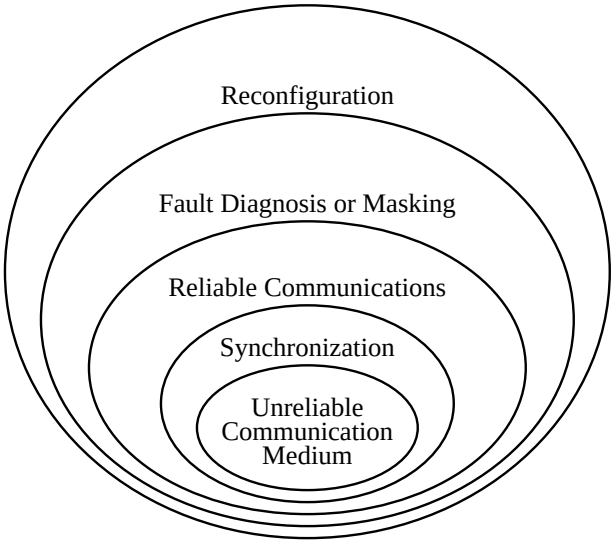

Figure 2: The consensus-based kernel of a responsive operating system.

3 UNICON Specification

We assume that all of the processors in the system are logically completely connected by an underlying communication medium. Moreover, there exists a distribution of message delays such that a processor may estimate the time that a message was sent from the time it was received and from which processor it was sent [Cristian 1989]. We assume that there exist mechanisms for scheduling and preempting tasks on a processor and that all processor clocks have been synchronized with UNICON as described in Section 5.2. We also assume that the number of faults are bounded according to the fault model employed.

For consensus to take place, at least five questions must be answered. First, who is to be involved in the consensus? The answer to this would be a list of processors. Second, what are the characteristics of those involved? That is, the fault models of the processors are needed. Third, what is to be agreed upon? The consensus task must know what information is to be collected. Fourth, what is the desired timeliness of the task? Fifth, what structure should the consensus take? In other words, do all the processors involved need all of the consensus results or will a subset of them be sufficient.

The structure of the consensus follows the taxonomy given in Fig. 3. Either it is a *application-specific* consensus or it is a *global* consensus, and for each of these possibilities, either it is a *partitioned* consensus or it is a *single-level* consensus.

An *application-specific* consensus targets only those processors involved in the consensus that are executing a particular application or some portion of an application. For example, consider a replicated file server implemented

via quorum reads and writes. For a read or write, a consensus of all processors involved in the replicated file server would be initiated, but based on the replicated file server's decision process only a quorum of those would actually perform the consensus. On the other hand, a *global* consensus is performed by all of the processors specified.

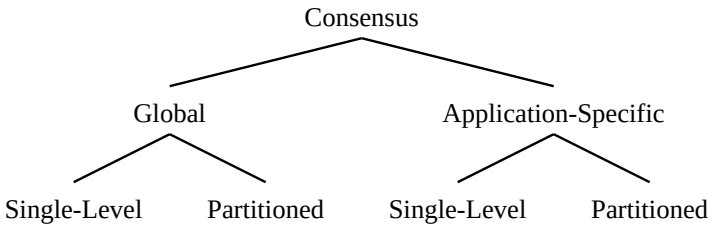

Figure 3: A taxonomy of consensus structures.

A consensus may be *partitioned* if there is an idea of importance attached to a processor's need for some part of the consensus. For example, consider the case of the system diagnosing itself. For a large system, it is very likely that some processors have very few dealings with many of the processors in the system, so it would be superfluous for it to know the diagnosis of those other processors. Therefore, the diagnosis consensus should be partitioned such that each set of processors that has an immediate need for each others diagnosis reaches consensus. These independent partitions then may be organized such that when a processor needs consensus information from outside of its partition it may receive it in short order. Such a technique called the Hierarchical Partitioning Method (HPM) was introduced in [Barborak and Malek 1992] and is assumed to be available to the UNICON process. In contrast, a *single-level* consensus completes with all the processors involved knowing all of the consensus results, i.e., the consensus is globally executed among the specified processors.

From the defining requirements for consensus, we envision the declaration of UNICON as

```
unicon(con_id *id,PE_set members,con_top topology,con_type type,con_time priority)
```

where `id` is set to the identification number of this instance of consensus and `members` is the set of processors involved in the consensus. We assume that the set `members` refers to a specific set of physical processors, but it could refer to a logical set maintained by the operating system. `topology` is a structure of two fields, namely, `structure`, which is evaluated from the set {`single-level, partitioned, unspecified`}, and `restriction`, which is evaluated from the set {`global, application-specific`}. `type` is an enumerated variable of a set including {`synchronization, configuration, diagnosis, communication`}. This set may be extended, but for the purposes of this paper, we use only four members. We may also consider the case in which `type` is a union of the various members of this set such as when multiple purposes may be served by a single instance of UNICON. For this paper we assume that each consensus has a distinct goal, but in practice, various information items could be combined in the consensus messages in order to perform multiple functions at once. `priority` determines the timeliness requirements of the responsive consensus.

From this simple system call, it is possible to invoke any number of the consensus protocols. By examining the fault models of the processors in `members` the system can choose a suitable consensus algorithm; by examining the number of processors in `members` the system can decide whether a partitioning method is useful or the structure may be specified with `topology`. The consensus algorithm is abstracted across synchronization, configuration, diagnosis, communication and consensus information requirements with the `type` variable.

`priority` defines the timeliness requirements of a particular instance of UNICON. It is a structure with members `time`, which is an absolute, real-time value as set by the synchronization process in Section 5.2, `periodic`, which is a boolean stating whether or not the consensus should be scheduled periodically, `duration`, which in the case that the consensus should be scheduled periodically is the length of that period, and `sched`, which is of the set {`urgent, deadline, asap, lazy`}. Note that specifying that the consensus should be completed in 10 seconds implies that `time` should be set to the current time plus 10 seconds. Also, if a consensus is periodic then after each deadline the `time` variable should be increased by `duration` and the task rescheduled. The meaning of these elements are as follows:

`urgent`: The consensus is to be completed by `time` even if this implies that all other scheduled tasks must be preempted. If `time` has passed, then the consensus task should be the sole purpose of the processor until its completion.

`deadline`: The consensus should be scheduled during the spare capacity of the system to complete by `time`. If `time` passes before completion, then the incomplete results should be provided.

`asap`: The consensus should be begun and completed as soon after `time` as the processor's schedule allows. By analogy to an interrupt hierarchy, it could be called a polite consensus.

`lazy`: The consensus should be scheduled during the processor's spare capacity, but if it is not completed by `time` then it should be abandoned with the incomplete results reported.

As we assume that all processor clocks are synchronized to within some bound called `synch_error`, `time` has meaning, although somewhat inconsistent, throughout the consensus. When we initiate UNICON, therefore, it is necessary to let other processes know that a result is needed by `time - synch_error` in order that the result will be available at the initiating process by `time`.

4 UNICON Implementation

The `unicon` task is responsible for initiating UNICON. The processor that it runs on may or may not be an element of `members`. `unicon` represents the interface between a user, i.e., application software, or the operating system and the actual consensus mechanism. As such, it accepts a very broad description of what is expected and produces enough detail to actually achieve a correct result. A detailed description of `unicon` follows.

```
unicon(con_id *id,PE_set members,con_top topology,con_type type,con_time priority)
{
   con_data table_contents;

   *id = next_con_id();
   if(topology.structure == unspecified)
      if(single-level_msgs(members) > partitioned_msgs(members))
         topology.structure = partitioned;
      else
         topology.structure = single-level;
   priority.time = priority.time - synch_error;
   multicast(members,``con_start'',*id,members,type,topology,priority);
}
```

The variable `id` is set such that at any particular time, a consensus task, whether executing or completed, may be identified by its unique consensus identification number. This allows a single processor to execute many consensus jobs concurrently. Accessing the consensus information would be done via the identification number through a system call that would either return this information or report that the consensus is incomplete. Finally, the `topology` variable holds information on how the consensus members should coordinate, i.e., in a single-level or partitioned style, in order to minimize message costs and execution time. After calculating the proper values for these variables, `unicon` sends messages by way of a multicast to the members of the consensus with the information they need to proceed. The multicast is assumed to be unreliable, i.e., a failure of the initiator or the communication medium could result in none or only some of the addressed processors receiving the message.

In order to schedule `unicon`, a processing element must have an intimate knowledge of the processors involved in the consensus. We look more closely at this problem as well as scheduling the rest of the related tasks in Section 5.1.

Upon receiving a message from `unicon`, a processor schedules `con_task` to take the required actions according to the value of `priority` in the message. For example, if the message is part of an urgent consensus whose deadline has already passed then `con_task` will begin immediately. This resident consensus task must be able to start a consensus, query other processors for consensus data, receive and process consensus data and if running on the initiating processor, receive reports upon completion of the consensus. Below is a description of the `con_task`, `con_report` and `con_start` routines.

```
con_task(msg_contents msg)
{
```

```
   switch(head(msg))
   {
      ``con_start'':
         con_start(tail(msg));
         break;
      ``con_query'':
         con_query(tail(msg));
         break;
      ``con_data'':
         con_data(tail(msg));
         break;
      ``con_report'':
         con_report(tail(msg));
         break;
   }
}
```

In `con_report`, the function `list` keeps track of who has reported while the function `sufficient` returns a true value if enough reports have been received to produce a valid consensus table. For an example of how `sufficient` might work, see [Kreutzer and Hakimi 1988]. `whence_report` is the processor sending the report.

```
con_report(con_id id,con_data table)
{
   if(¬ sufficient(id,list(id)))
   {
      list(id) = list(id) ∪ whence_report;
      table_contents(id) = table_contents(id) ∪ table;
   }
}
```

In `con_start`, `initiator` represents the processor on which the original `unicon` was started. If it is determined that an application-specific consensus does not affect the processor running `con_start` then the processor exits the consensus process.

```
con_start(con_id id,PE_set members,con_type type,con_top topology,con_time priority)
```

```
{
   if(topology.restriction == application-specific)
      if(me ∉ application)
         return;
   allocate_table(id,members);
   consensus(members,id,type,topology,priority);
   sendmsg(initiator,``con_report'',id,table_contents(id));
}
```

Processor	Messages
A	—
B	$val(B) = 0, val(C) = 1, val(D) = 0$
C	$val(B) = 1, val(C) = 1, val(D) = 0$
D	$val(B) = 0, val(C) = 0, val(D) = 0$

Figure 4: A completed consensus table for a Byzantine agreement.

The `con_start` routine is responsible for creating and maintaining the consensus data table, identified by `id`, that contains all the data received by other members of the consensus except for redundant information. For example, Fig. 4 shows the completed consensus table for processor A of a four-member, Byzantine agreement in which processor C is faulty. In this example, each processor sent its value to each other processor in the initial round, e.g., A sent $val(A)$ to B, C and D. Then each processor sent the values received in the first round to every other processor, e.g., B sent $val(C)$ and $val(D)$ to A. From this information depicted in Fig. 4 and using a majority vote, A knows that all fault-free processors will agree that $val(B) = 0$, $val(C) = 1$ and $val(D) = 0$ [Pease *et al.* 1980].

By combining the information in Fig. 4 with information about the fault models of processors B, C and D, processor A can determine what information is needed to complete the consensus as well as when that completion occurs. It is up to the algorithm `consensus` to determine when processors should be queried in order to reach this completion. If all of `members` behave according to the same fault model, then any of the procedures mentioned in [Barborak *et al.* 1991] along with the HPM techniques should be sufficient for `consensus`. On the other hand, if the fault models are different, then the table must be maintained somewhat uniquely.

Fig. 5 shows one view of processor fault models and their relationships to each other. Each class is a subset of the class that is listed next, i.e., omission faults include crash faults and Byzantine faults include timing faults. Below are descriptions of these models.

crash fault: The fault that occurs when a processor loses its internal state or halts. For example, a PE that has had the contents of its instruction pipeline corrupted, or has lost all power has suffered a *crash fault*.

omission fault: The fault that occurs when a processor fails to meet a deadline or begin a task.

timing fault: The fault that occurs when a processor completes a task either before or after its specified time frame. This is sometimes called a *performance fault*.

Byzantine fault: An arbitrary fault such as when one processor sends differing messages during a broadcast to its neighbors. More generally, this is every fault considered in the system model.

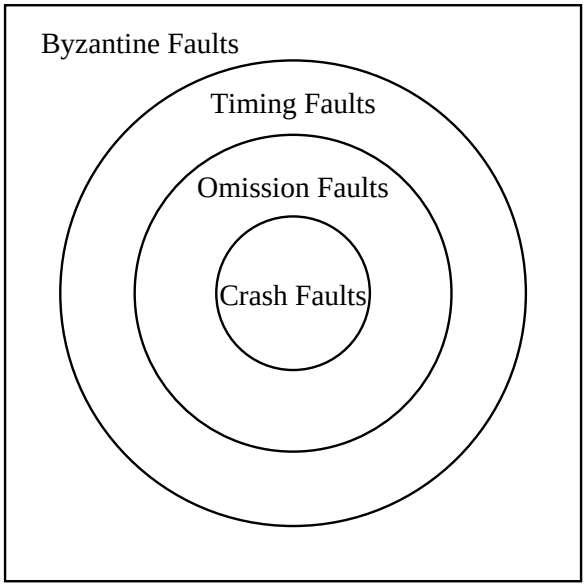

Figure 5: A fault classification.

Using this classification of fault models, consider the case of managing the following consensus table of processor A appended with the fault class of each processor in the consensus.

Processor	Fault Model	Messages
A	—	—
B	crash failures	$val(B) = 0, val(C) = 1$
C	Byzantine failures	$empty$
D	timing failures	$val(D) = 0$

Figure 6: A completed consensus table for processors of various fault models.

The table in Fig. 6 is smaller than the table in Fig. 4 as a direct result of taking advantage of the fault models of processors B and D. As B suffers only from crash failures, it follows that if a message is received from B then it is fault free. If no message is received, then B must have crashed. In other words, the status of B may be determined directly from B and therefore C and D need not relay any information about B to processor A. Similarly, the status of D may be determined directly from D as long as the system is synchronized and message arrival times may be checked with expected arrival times. Processor C on the other hand suffers from Byzantine failures and therefore cannot be relied upon to accurately portray its status. Yet B, for example, may be counted on to correctly convey to all other PEs whatever value C sent it. Therefore it is sufficient for C to send a message to a fault-free, non-Byzantine-failing processor that in turn broadcasts the value to the other members in the consensus. In Table 6, B has been chosen for this task. As in the previous example, A knows that all fault-free processors will agree that $val(B) = 0$, $val(C) = 1$ and $val(D) = 0$.

Assuming that each processor in `members` suffers from one of the failure semantics in Fig. 5 and that a single-level topology for the consensus algorithm is employed, `consensus` may be as follows where `Byzantine` is the subset of processors in `members` that are Byzantine failing. `me` is the processor running the particular instance of `consensus` and `val(me,type)` returns data according to the type of consensus being performed. `wait_entry`, which waits for an entry in the consensus table to be filled, will timeout if the time needed for a fault-free processor to deliver the required message has been exceeded. If `entry` is called for a non-existent consensus table entry, e.g., when a processor has failed to send a message, then the value $\perp$ is returned. If $\perp$ is received as the value of a processor, though, it is stored as $default\_value$ to distinguish the case when a Byzantine processor fails to send a message and a non-Byzantine processor fails to forward a message. The set of processors in `members` are ordered by their identification numbers which are returned by the function `PE_id`, and once again, the multicast function is unreliable.

```
consensus(PE_set members,con_id id,con_type type,con_top topology,con_time priority)
{
   PE q,q',r;

   q = q_i | ∀ q_i,q_j ∉ Byzantine :  i ≠ j, PE_id(q_i) > PE_id(q_j);
   if(q == ∅)
   {
      byzantine_agreement(members,id,type,topology,priority);
      return;
   }
   if(me ∉ Byzantine)
      multicast(members,``con_data'',id,val(me,type),priority);
   else
```

```
      sendmsg(q,``con_data'',id,val(me,type),priority);
   if(me == q)
      ∀ r | r ∈ Byzantine
      {
         wait_entry(id,r);
         multicast(members,``con_data'',id,entry(id,r),priority);
      }
   while(¬ table_complete(id))
   {
      if(me ∉ Byzantine)
         ∀ r | r ∉ Byzantine, ⊥ == entry(id,r)
         {
            sendmsg(r,``con_query'',id,type,idata,priority);
            wait_entry(id,r);
         }
      q' = q_i | ∀ q_i,q_j ∉ Byzantine :  i ≠ j, PE_id(q) > PE_id(q_i) > PE_id(q_j);
      q = q';
      if(q == ∅)
         break;
      if(me == q)
         ∀ r, wait_entry(id,r) | r ∈ Byzantine
            if (⊥ == entry(id,r))
            {
               sendmsg(r,``con_query'',id,type,idata,priority);
               wait_entry(id,r);
               multicast(members,``con_data'',id,entry(id,r),priority);
            }
   }
   if(¬ table_complete(id))
      byzantine_agreement(members,id,type,topology,priority);
}
```

consensus is based on the notion that a testably fault-free processor such as a crash-, omission- or timing-failing processor can relay a message from a Byzantine-failing processor to avoid the discrepancies caused by allowing a faulty Byzantine processor to broadcast its data. Therefore, in **consensus**, the first step is to determine which

processor q will act as the consistent liaison with the Byzantine population. If there is no q then consensus degenerates into Byzantine agreement. Second, a processor broadcasts its data if it is not Byzantine otherwise it sends its data to a non-Byzantine q. Third, q broadcasts the values it received from the Byzantine processors regardless of whether they are true or false as we are only concerned with agreement. If q remains fault free and the non-Byzantine processors successfully transmit their data then consensus is done in two rounds of message passing, but if it fails then another non-Byzantine processor must take over. If all non-Byzantine processors fail before completion, then the algorithm again degenerates into Byzantine agreement.

In order to abstract the consensus process, consensus does not request specific data, but simply data qualified with the variable type set in the unicon initiation. This is a result of the observation that the consensus task has no use for the actual data in the consensus table, but only for the fact that data has been received from processors behaving under various fault models. Therefore, all queries to other processors are for generic data which is only given meaning when con_query illicits the exact values for the data from its host processor and when the final consensus information is used. Therefore, in the following description of con_query, clock returns the processor's time for synchronization purposes, local_config returns what processors and communication channels are directly connected to the processor, test accepts input data and returns some value for examination by other processors during diagnosis, variable accepts some input data to return a portion of a local process' state and table_entry uses a consensus id and some input data to return a subset of the information held in one of the processor's consensus tables.

```
con_query(con_id id,con_type type,input_data idata,con_time priority)
{
   switch(type)
   {
      synchronization :
         sendmsg(whence_query,``con_data'',id,clock(),priority);
         break;
      configuration :
         sendmsg(whence_query,``con_data'',id,local_config(),priority);
         break;
      diagnosis :
         sendmsg(whence_query,``con_data'',id,test(idata),priority);
         break;
      communication :
         sendmsg(whence_query,``con_data'',id,variable(idata),priority);
         break;
      consensus :
```

```
            sendmsg(whence_query,``con_data'',id,table_entry(id,idata),priority);
            break;
      }
}
```

The final task is `con_data` which simply receives data, checks that it is not redundant and if not, stores the data in the consensus table. The function `type` returns the type of the consensus identified by `id`.

```
con_data(con_id id,input_data idata)
{
   if(¬ redundant_entry(id,whence_report,idata)
   {
      if(type(id) == synchronization)
         idata = idata ∪ clock();
      entry(id,whence_report) = entry(id,whence_report) ∪ idata;
   }
}
```

5 UNICON Applications

5.1 A Time Analysis For Scheduling

The algorithm we have given is proposed as the basis of a responsive system. As such it must be timely. In this section, we examine the proposed algorithm from the perspective of the scheduler that must decide when to start the task and whether or not it can possibly complete before the given deadline.

On the initiating processor, the problem of scheduling is primarily a problem of determining the latest time at which `unicon` may be started and still complete before `time`. Note that if `priority` of `unicon` is either `asap` or `lazy` then the scheduling policy is already determined as discussed in Section 4. The calculations in `unicon` are insignificant compared to the time that will be spent waiting for `members` to actually reach a consensus, therefore, this analysis will focus on the messages required to complete.

Assume that the period starting when `unicon` initiates its multicast of `con_start` messages and ending when `sufficient` becomes true is called `con_duration`. Then for the scheduler to meet the deadline `time` it must begin `unicon` by `time - con_duration`. If the task is of the type `urgent` then it might be necessary to preempt tasks to begin by this time. If `time` has already passed and `unicon` is of the type `deadline` or `lazy` then `unicon` need not be scheduled at all.

Determining con_duration is difficult as the value will depend on the fault models of the processors involved, whether or not any of the processors are faulty or become faulty during the consensus, which algorithm is being used for the task consensus and the system characteristics such as the type of communication network being used. The best that we can hope for is a worst case estimate or an estimate that will be true with a certain probability. For the analysis, we assume that the longest time a message requires to traverse any communication link including processor transfer and setup times is δ. So, for example, in a point-to-point network of diameter k, the longest time required to broadcast a message is $k\delta$. We denote the distance (in terms of the minimum number of links between them) between two processors as $\mathtt{d}(\mathtt{p}_i,\mathtt{p}_j)$.

The first delay is the multicast of the con_start messages in unicon. The time required to complete this stage is δmax_d where max_d is defined as

$$\mathtt{max\_d} = \mathtt{d(initiator,p)} \mid \forall\ \mathtt{p,q} \in \mathtt{members},\ \mathtt{d(initiator,p)} \geq \mathtt{d(initiator,q)}$$

In other words, δmax_d is the longest time required to send a message between initiator, i.e., the processor running the unicon process, and any of the processors in members.

The next delay is to wait for members to perform the task consensus and report back to initiator. In the case that the consensus is of the type urgent or deadline, its start may be delayed by each of the members p so that it is completed by time - δd(initiator,p). That is, each processor may schedule consensus such that it completes just in time to transmit its report back to initiator such that initiator will receive the report just at the deadline of unicon. Let con_min be the minimum time a processor must allot for consensus. Therefore, each of members must start consensus by time - con_min - δd(initiator,p).

con_min is highly dependent on the consensus algorithm being employed. For the processor membership protocols based on processors that suffer from crash-, omission- and timing-failures, Cristian has given a detailed time analysis in [Cristian 1991]. Basically, the problem reduces to determining the maximum number of messages that any processor must request and then receive in order to complete, that is, the number of serial rounds of message passing required. Suppose that there are at most j rounds in a particular implementation despite failures of processors. Then we can say that

$$\mathtt{con\_min} \leq 2j\delta\mathtt{diameter(members)}$$

The factor of two has been added to accommodate the case when replies in a round cannot be performed in parallel with requests in the next round.

The algorithm we gave, consensus, is somewhat worse than Byzantine agreement in the worst possible scenario. In this case, the non-Byzantine processors fail one after the other until there are only Byzantine processors remaining, resulting in the default of a Byzantine agreement algorithm. Assuming that there were i non-Byzantine processors, the delay before initiating Byzantine agreement would be $i\delta$diameter(members) as each processor timed-out. (Note that δd(p,q) is the timeout period that q must wait before a message from p is considered lost and that d(p,q) $\leq$ diameter(members).) On the other hand, if at least one non-Byzantine processor

survives, consensus will be completed in two rounds. Let a be the probability that a processor will fail during consensus. Then, the probability that at least one processor survives and that consensus will be completed in 2δ`diameter(members)` is $(1 - a^i)$. If this probability is sufficiently high to meet the responsiveness of the system then `con_min` can be set at this lower value.

The final delay is simply the time needed for `members` to report back to `unicon` which is simply δ`max_d`.

Finally, we can give an estimate for `con_duration` where `consensus` is implemented as shown in Section 4 and the probability that all non-Byzantine PEs will fail is negligible.

```
con_duration = 2δ[max_d + diameter(members)]
```

Under these assumptions then, the scheduler on `initiator` must begin the task `unicon` at `time - con_duration`.

5.2 Clock Synchronization

Clock synchronization is simply a manifestation of the general consensus task, and as such, UNICON is sufficient to perform it. In this section, we shall show how `unicon` may be used to synchronize the system's clocks which we assume are reliable and run at equal rates. We shall assume that the processors are either crash-, omission- or timing-failing and that, as in Section 5.1, the time for a correct processor `p` to send a message to a correct processor `q` is less than or equal to δ`d(p,q)`. From here, the process is straightforward.

Some processor `initiator` begins the task `unicon(&id,members,topology,synchronization,priority)` where priority is set according to the need of this synchronization. The elements of `members` receive the instructions to begin a consensus and tables of values specified by `type`, in this case equal to `synchronization`, are formed in which each entry is a clock value of a processor `q`, denoted `M(q)`, timestamped with when the message carrying this data was received, denoted `recd(M(q))`. Let `p` be the processor building a particular table and `C(p)` be the current clock value of `p`. Then, `p` knows that `C(q) = M(p) + (C(p) - recd(M(p)))` $\pm$ δ`d(p,q)`$/2$. Putting a lower bound on transmission time or taking a probabilistic approach as in [Cristian 1989] can make the estimation of `C(q)` more accurate. Now the processors of `members` have consistent consensus tables within δ`diameter(members)`$/2$ of each other and synchronization may take a number of routes including choosing the average, median, maximum or minimum time for each processor.

5.3 Reliable Communication

A reliable multicast is a communication from one PE to many in which it is guaranteed that the outcome is predictable in the presence of a bounded number of faults for a particular fault model and for bounded-time message passing. For a fail-stop model, the result is if at least one fault-free, addressed PE receives the message then all of the fault-free, addressed PEs will receive the message. One approach to this problem is to initiate a consensus among the addressed PEs and the sender for them to decide on the value of the multicast.

For UNICON, `members` is the group of PEs that should receive the multicast as well as the sender of the multicast and `type` is set to `communication`. The sender of the message begins the task `unicon(&id,members,topology, communication,priority)`. An unreliable multicast spreads the consensus to the set `members`. At this point the details of `consensus` dictate the exact process, i.e., the multicast may proceed based on masking or fault diagnosis. Assuming that `consensus` is implemented as described in Section 4 and that the set `Byzantine` is empty, the consensus continues until each PE has completed its consensus table, that is, until each has heard from every fault-free processor in `members`. Because each processor is testable (`Byzantine` = $\emptyset$), as would be the case if `consensus` were based on the ideas of system diagnosis, it is possible for a piece of data to propagate to the fault-free processors in `members` and not be masked out. Therefore, a consensus table is complete whenever a processor receives the communication data from a fault-free PE, and a fault-free PE will continue consensus without having heard the communication data until it decides that it has communicated with all of the fault-free elements of `members`. Thus we may achieve the reliable multicast described above for fail-stop processors.

6 Conclusion

We have proposed a universal consensus protocol called UNICON that implements consensus as a truly general task in the kernel of a distributed operating system. Rather than approaching synchronization, reliable communication, diagnosis, configuration and other distributed tasks as separate, unrelated jobs, UNICON treats them as the same job only differentiating themselves at the point where actual data is read from the processors and stored in the consensus table. In terms of system design, the result is that only one algorithm needs to be designed and maintained as opposed to four or more resulting in better reliability and efficiency without a loss of functionality. UNICON allows a significant amount of flexibility as far as the scope of a particular instance of consensus is concerned as well as in terms of the failure semantics of the processors involved.

We presented a simple consensus algorithm showing how processor fault models may be exploited to reduce message costs and therefore time to complete. The HPM allows for a heterogeneous system in that each partition may behave according to different assumptions, but it must still be expected that the occasional, single-level consensus will have disparate members. Therefore our algorithm adapts its queries about other processors according to the fault models of those processors.

Next we analyzed UNICON in terms of the time it requires and what limitations this places on scheduling the task. We looked at this problem from the viewpoint of the scheduler that initiates UNICON as well as the schedulers on the various processors actually performing the consensus algorithm.

Finally, we showed how UNICON can be used to perform a simple synchronization of the system's clocks and reliable communications.

We anticipate that UNICON will become a foundation for responsive operating systems of the future in which demanding requirements for fault tolerance and timeliness are expected to be met.

References

[Barborak *et al.* 1991] M. Barborak, A. Dahbura, M. Malek, "The Consensus Problem in Fault-Tolerant Computing," Technical Report TR-91-40, Department of Computer Sciences, University of Texas at Austin, November 1991.

[Barborak and Malek 1992] M. Barborak, M. Malek, "Partitioning for Efficient Consensus," *to appear in the Proceedings of the Hawaii International Conference on Systems Sciences 1993.*

[Cristian 1989] F. Cristian, "Probabilistic Clock Synchronization," *Distributed Computing* Vol. 3, pp. 146-158, 1989.

[Cristian 1991] F. Cristian, "Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems," *Distributed Computing* Vol. 4, pp. 175-187, 1991.

[Kreutzer and Hakimi 1988] S. Kreutzer, S. Hakimi, "Distributed Diagnosis and the System User," *IEEE Transactions on Computers*, Vol. 37, No. 1, pp. 71-78, January 1988.

[Malek 1991] M. Malek, "Responsive Systems: A Marriage between Real Time and Fault Tolerance," *Fifth International Symposium on Fault-Tolerant Computing Systems*, Nürenberg, pp. 1-17, 1991.

[Malek 1992] M. Malek, "A Consensus-Based Framework for Responsive Computer System Design," *Proceedings of the NATO Advanced Study Institute on Real-Time Systems*, Springer-Verlag, St. Martin, West Indies, October 5-18, 1992.

[Pease *et al.* 1980] M. Pease, R. Shostak, L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, Vol. 27, No. 2, pp. 228-234, April 1980.

[Schneider 1984] F. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," *ACM Transactions on Computer Systems*, Vol. 2, No. 2, pp. 145-154, May 1984.

[Valiant 1990] L. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, Vol. 33, No. 8, pp. 103-111, August 1990.