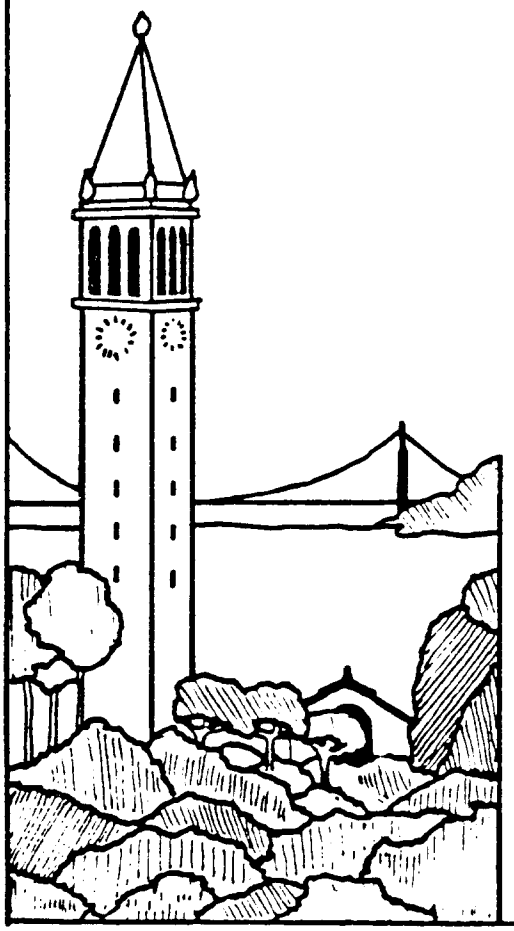
THE NETWORK EVENT MANAGER

Yih-Farn Chen, Atul Prakash,
C. V. Ramamoorthy



Report No. UCB/CSD 86/299

June 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

THE NETWORK EVENT MANAGER

Yih-Farn Chen
Atul Prakash
C. V. Ramamoorthy

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

ABSTRACT

In order to utilize network resources effectively and be resilient to network failures, network clients must adapt to dynamic network state. However, detecting and handling asynchronous network events, which change network state, is a complicated task. Instead of imposing this burden on each network client, a *Network Event Manager* (NEM) accepts service requests of the form { *event, action* } from the clients. It maintains network status, performs event detection, and executes actions on behalf of a client when the client's registered event occurs. This paper discusses the issues involved in designing and implementing a network event manager. A network event manager can be used as a user watchdog or be applied to the adaptive control of *pulsating computation,* which is a single-phase or multi-phase distributed computation structured as a set of master and slave processes. We show that many time-consuming computations, like compilation, typesetting, and computer animation, can be structured as pulsating computations. The coupling of a shared network file system and a network event manager simplifies the reconfiguration of pulsating computations with changes in network status. Experiences with two prototype implementations of Event Manager are described.

1. INTRODUCTION

In order to utilize network resources effectively and to guard against various network failures, a distributed computation should adapt to changes in network status. However, it requires nontrivial effort to collect information about the status of individual entities in a network and make assertions about the global network status. For example, to detect that a token is missing in a token ring or that a network partition has occurred in a point-to-point network requires many message exchanges among network nodes.

Since many distributed computations rely on the availability of status information, we advocate a layered approach for structuring computations that depend on the network status: (Figure 1): The goal of this layering approach is to make it easier for an application process to obtain the network status. Each layer reduces the visibility of the non-determinism present in the network to the upper layers.

This work is supported in part by Ballistic Missile Defense under contract No. BMDSC DAS660-85-C-0029 (1-482427-25545).

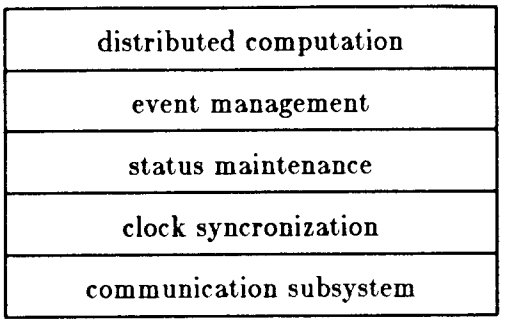


Figure 1. A layering approach for sharing dynamic network status

The first layer is the communication subsystem. It guarantees message delivery and the ordering of messages between non-faulty nodes.

The second layer is the clock synchronization layer. It guarantees that clocks of all non-faulty nodes are close to real time and observe the local causality property[1], and the difference between any two non-faulty clocks is less than a known value δ.

The third layer is the status maintenance layer. It is responsible for maintaining the status of distributed objects like nodes, links, users, or whatever are required by the upper layer. It is realized by a protocol in which each node sends the status of its local objects and communication links periodically to a distributed or centralized network status maintainer. Each status report is timestamped using the local clock time. The network status maintainer makes the status information available to other network nodes either through broadcasting or through a shared network file system.

The fourth layer is the event management layer. A network event manager accepts service requests from the network clients (processes or users). A request is of the form <event, action>. When a registered event happens, the corresponding action is taken on behalf of the client. The action can be as simple as sending a notification to the client or as complex as initiating a process at a specified node. The event manager detects events by analyzing the network status information provided by the status maintainer. The paradigm used in the network event manager is outlined in Figure 2. This paper examines various issues in the design and implementation of this layer.

The fifth layer is the distributed computation layer. Based on the information provided by the event manager, a distributed computation can be restructured to adapt to the changing network status.

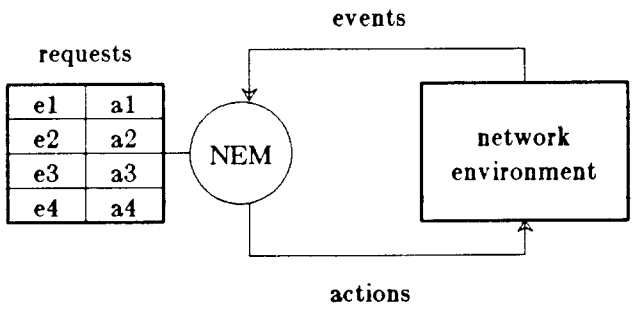


Figure 2. The Network Event Manager (NEM)

The idea of event management is not new in a local environment. Program debuggers provide an example of simple event management. Events in a debugger are calling of a procedure, changing the value of a variable, execution of an instruction, etc. The user can post a request with

the debugger to suspend processing when these types of events occur or to print out the values of certain variables. Another example of this paradigm is the *at* command in Unix†. It allows a user to post a request with the system to execute any command at a specified time. The *notifier* of the SunView‡ system also adopts an event-triggered approach. The notifier receives all asynchronous events (window entering/exiting, button pressing, etc.) affecting a user process, and synchronously dispatches these events in the right order to the clients.

However, to apply the idea of event management in a distributed environment is a challenging task. The global status must be constructed in *real time* from partial views (timstamped with local clocks), supplied by individual nodes. Delays caused by message transmission can affect event detection time and action firing time. Event ordering and action ordering becomes more difficult than in a local environment. This report proposes an approach to solving this problem.

The network event manager we proposed here can be used to implement a distributed *at* command or to act as a general *user watchdog* that handles requests like the following:

(1) When the load average of *ucbernie* is lower than 1.0 and the number of users is less than 10, execute the command *ditroff -ms -Pip paper*.

(2) When more than half of the workstations are idle, execute an *animation worm*[3] on all the idle workstations.

(3) When the owner of workstation A comes back, kills my compilation jobs on that workstation and restarts it on workstation B.

We postpone the discussion on how these events can be represented to Section 4.1.

The network event manager can also be used to support distributed computations by taking care of the detection of asynchronous events. We studied this problem in a specific context — a local area network with a shared file system. We discovered that a class of distributed computations, namely the *pulsating computations*, can best exploit the advantages of the shared file system if the network event manager is available. We discuss this problem in detail in Section 6.

We have implemented two prototype event managers and proved the feasibility of the idea. One implementation employs a Lisp pattern matcher to perform the event matching task; the Lisp process communicates with a C process to obtain the network status. The other implementation employs pure C data structures and routines for event matching. We discuss the details of these two implementations and various tradeoffs in Section 5.

The rest of this paper is organized as follows. Section 2 gives some background information on the clock synchronization problem. Section 3 briefly discusses the status maintenance problem. Section 4 describes the design issues involved in building a network event manager. Section 5 describes two prototype implementations. Section 6 describes how the network event manager can be used to support pulsating computations. Section 7 discusses potential applications of the network event manager, followed by the conclusion in Section 8.

2. THE CLOCK SYNCHRONIZATION LAYER

The problem of clock synchronization and ordering of events in a distributed environment is discussed by Lamport in his classic paper[1]. Clock synchronization algorithms can be compared based on the following criteria (under the same assumptions on communication subsystems):

C1. Local causality

C2. Global causality

C3. Maximum difference between any two clocks

C4. Closeness to real time

C5. Degree of fault tolerance

† Unix is a registered trademark of Bell Laboratories.

‡ SunView is a registered trademark of the Sun Microsystems, Inc.

The distributed algorithm proposed by Lamport[1], satisfies the local causality property and the global causality property, but does not work in case of process failures or communication failures. The algorithm does not guarantee closeness to real time either; actually, all the clocks drift further away from real time as the phases of the algorithm continue. The Berkeley TEMPO [4] algorithm uses a mater-slave control structure with an election algorithm to select a new master when the master fails[5]. It satisfies the local causality property and the clocks are kept within 40 milliseconds (in the particular network experimented) from each other if the master does not fail. The authors of TEMPO also claim that the algorithm keeps the clocks close to real time. For clock synchronization algorithms that deals with arbitrary process or clock failures, please refer to [6-9].

The clock synchronization layer we proposed should satisfy the following requirements:

CR1. Local causality: If event a on a node happens before event b on the same node, it should have a timestamp smaller than that of the event b.

CR2. Close synchrony: Maximum difference between any two non-faulty clocks should be less than δ. How small δ should be depends on the applications. The close synchrony is necessary for combining timestamped status reports from different nodes in a reasonable way.

CR3. Closeness to real time: This is necessary because a user external to the distributed system may use the real time as a reference in his requests to the event manager. To simplify our discussion in the following section, we assume that the real time clock runs on a virtual node, so that all the clocks, including the real time clock, satisfy CR2.

We do not consider the global causality property necessary for our clock synchronization layer; we feel that it would be too expensive to implement and our status maintenance layer does not require that. The degree of fault tolerance (whether n-processor failures, two-faced clocks[7], network partitioning, etc., are tolerated) of the clock synchronization algorithm depends on the application environment. For example, in a university environment, we can simply use an algorithm similar to TEMPO; when the master clock fails, the status maintenance service and the event management service may be discontinued or degraded until a new master is elected and clocks are resynchronized. In other environments where the status maintenance service can not be interrupted, the clock synchronization algorithm needs to be more involved. For example, a new clock synchronization algorithm proposed by Anupam and Ramamoorthy satisfies CR1, CR2, and CR3 with greater accuracy in a Byzantine environment[9].

3. THE STATUS MAINTENANCE LAYER

At this layer, processors cooperate to collect the status of the network. To represent the network status formally, we can view a computer network as a collection of objects where each object has a set of attributes. Table 1 shows a simple abstract view of a computer network.

Object Type	Attributes
host	status (up/down), clock value, number of users, load average, up time
link	status (up/down)
user	ttyid, idle time, login status (on/off), hostname
device	status (running/idle/broken), queue length

Table 1. An abstract view of a computer network

The degree of elaboration of this abstract view affects the complexity and performance of the status collection process. Note that the objects listed in Table 1 are of interest to most network clients. We do not incorporate process objects in the abstract view, because collecting the status of all processes on a network is very expensive. However, for the purpose of process control and management, a user may need on demand services to maintain the status of processes created and

distributed by him in the network environment. Cabrera, Sechrest, and Cacéres suggest that a personal process manager (PPM) can be used for this purpose[10, 11].

The task of collecting the status of interesting objects can be *partitioned, replicated,* or both. If the task is partitioned, then each attribute value is sampled only by a single process. If the task is replicated, then an attribute value can be sampled by more than one process and a *consensus* algorithm may be necessary to reach an agreement on the attribute value. For example, the status of a link can be sampled by its neighboring hosts, then the results are passed to the rest of the network. Each host can then compute the status of that link based on the results collected.

Each status report should be timestamped using the local clock. For the purpose of event management, several status reports sampled at different times for the same attribute may have to be combined in order to detect whether an assertion (see Section 4.1) is true for a certain period. For example, the following request may be placed:

> event: The load average of ucbernie is less than 2.0 for 5 minutes
> action: start a typesetting process on ucbernie

To detect that event, we need to construct an approximation of the changes of the load average on ernie over time. In general, for integer-valued attributes, the zeroth order approximation (the dotted line) shown in Figure 3 can be used. For real-valued attributes, the first order approximation shown in Figure 3 or higher order approximations can be used. In either case, increasing the sampling rate provides higher accuracy for the detection of certain events, but incurs more overhead. The sampling rate should be individually decided based on the nature of each attribute. For example, the number of users on a host does not change so frequently as the one-minute load average of a host. Therefore, the sampling rate for the former should be smaller than the latter.

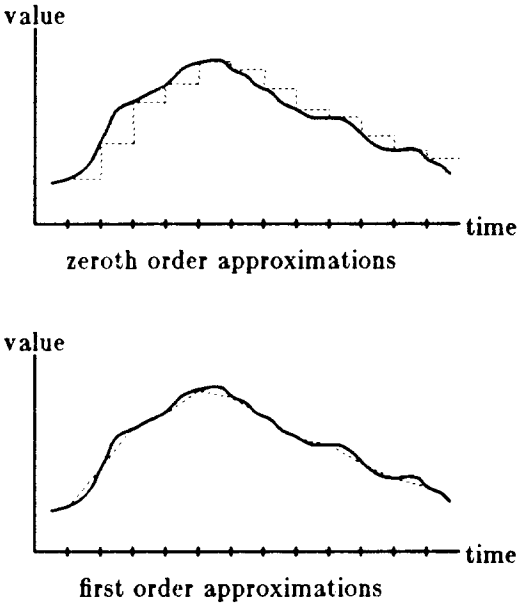


Figure 3. Approximating an attribute value as a function of time

Based on the above discussion, we can view the network status database as the one shown in Figure 4. It can be divided into two parts, the *current database* and the *history database.* The value of each attribute is sampled at different times. Before a new sample becomes available, each attribute assumes the value of the latest sample. When a new sample is obtained, the old one becomes part of the history database. The current values of all attributes form the current database. How "deep" the history database should be depends on the application environment. Note that many existing status maintenance schemes do not keep the history database.

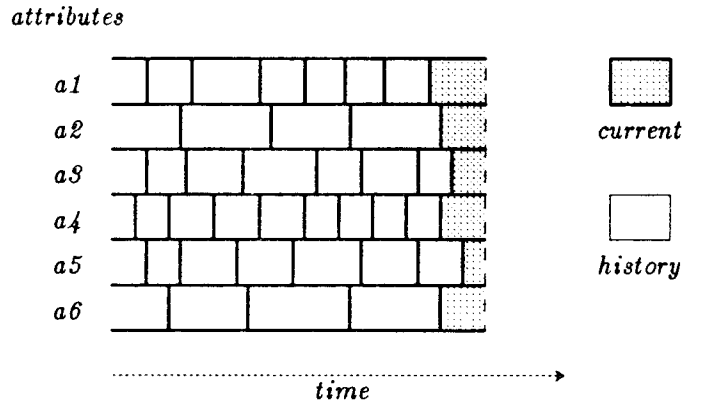


Figure 4. The network status database

We assume that the status maintenance layer provides the status database to the event manager; how this can be achieved is not the focus of this paper. However, we can give an example: the one that constructs the status database for the *ruptime* command on the Berkeley network where each host broadcasts the attribute values to the network every minute. Sampling rate is the same for all the attributes. A host is considered down if its status reports has not been received for 11 minutes. The messages are collected and the current database constructed independently by each host. Using the *ruptime* command, a user can view the current database at any time. No history database is available; however, an application process can construct its own history database by periodically sampling and saving the current database. Crash-failures are assumed for each host, i.e., no host will generate erroneous status reports at any case.

Note that the status collection problem we described here has a different flavor from the one described by Chandy and Lamport in their paper on *Distributed Snapshot*[12]. The distributed snapshot algorithm is appropriate for detecting stable properties; once a stable property becomes true it remains true thereafter. The algorithm may take arbitrarily long time to terminate and it does not depend on any clock synchronization scheme. Here we are trying to collect the dynamic network status *continuously*; the information must be provided in real time before it becomes invalid. A clock synchronization scheme is necessary so that combining timestamped status reports from different nodes can be meaningful.

4. THE EVENT MANAGEMENT LAYER

This layer is realized by a logically centralized server: the network event manager, which is responsible for accepting requests, matching events, firing actions, and managing requests. This section discusses the issues involved in event representation, request formation, and event matching.

4.1. Event Representation

An event is a change in the network status. To define it formally, we propose the concept of *simple event* and *complex event*. A simple event is a change of the attribute value of an object. The types of changes allowed depend on the nature of the attribute values. We distinguish among three types of events:

(1) *State Event*: an object has reached a particular state, e.g., up, down, idle, on, off, etc.

(2) *Differential Event:* the increase, decrease, or no change in the value of an attribute.

(3) *Assertion Event:* an event represented by the result of a logical comparison between an attribute value and a fixed value.

Complex events are logical combinations of simple events. Time and range information can be added to a simple or complex event to create two other types of events:

(4) *Time Event:* Any of the event types (1) through (3), or complex event, associated with a particular time is a time event.

(5) *Range Event:* Any of the event types (1) through (3), or complex event, associated with a period of time is a range event. The range can be specified in two ways: (type A) a starting time and an ending time, or (type B) the length of a period. How large the range can be depends on the depth of the history database. The interpretation is that the specified event must be true during the whole specified period.

Here are a few examples:

E1: If the number of users on ucbernie increases, ...
E2: If ucbernie is down, ...
E3: If the load average of ucbernie is larger than 5.0, ...
E4: If yfchen@ucbarpa is logged on at 1:05, ...
E5: If the load average of ucbarpa is less than 1.0 between 1pm and 2pm, ...
E6: If the number of users on ucbmurder is 0 for 10 minutes, ...

In the above examples, E1 is a differential event, E2 is a state event, E3 is an assertion event, E4 is a time event, E5 is a range event - type A, and E6 is a range event - type B.

A formal grammar can be specified (using the notation of Yacc[13]) for the definition of an event:

```
<event> : <complex_event>
     | <time_event>
     | <range_event>
     ;
<complex_event> : <simple_event>
          | (<complex_event> OR <complex_event>)
          | (<complex_event> AND <complex_event>)
          | (NOT <complex_event>)
          ;
<time_event> :  <complex_event> AT <time>
          | AT <time>
          ;
<range_event> : <complex_event> <range>;
<simple_event> : <state_event>
            | <differential_event>
            | <assertion_event>
            ;
<state_event> : <object> <attribute> <state>;
<differential_event> : <object> <attribute> INCREASES
               | <object> <attribute> DECREASES
               | <object> <attribute> DOES_NOT_CHANGE
               ;
<assertion_event> : <object> <attribute> <comparison_op> <value>;
<comparison_op> : > | < | == | >= | <=;
<range> : BETWEEN <time> AND <time>
     | FOR <period>
     ;
<object> : <object_type> <object_name>
```

Using this grammar, E1-E6 would be expressed as

E1: (host ucbernie #users INCREASES)
E2: (host ucbernie status down)
E3: (host ucbernie load > 5.0)
E4: (user yfchen@ucbarpa log_status on) AT 1:05
E5: (host ucbarpa load < 1.0) BETWEEN 1:00 AND 2:00
E6: (host ucbmurder #users 0) FOR 0:10

Examples of complex events are

E7: ((host ucbernie status up) AND (host ucbarpa status up))
E8: ((user prakash@ucbarpa log_status on) OR (user prakash@ucbernie log_status on))

It should be noted that a state event or an assertion event in the absence of time or range information refers to the current time (i.e. latest state known to the event manager). A differential event without time or range information refers to the two latest states known to the event manager. If the timing information is given, since all local clocks are sufficiently well synchronized (see CR2 in Section 2), the event manager assumes that any differences between them can be ignored when it combines status reports. Also, since all clocks are close to real time (CR3), a client can use real time as the reference time in a request. The approach is justified for many applications, including those that are discussed in this paper. For critical applications, the timing problem is more involved (see Section 7).

The above grammar cannot conveniently express all kinds of events, but it satisfy most needs. Our intention here is not to give a comprehensive grammar for events (that depends on applications), but to illustrate the concepts. Some example events that may be difficult to represent are

(1) The average number of users on host A, B, and C is greater than 10.

(2) The load average of host A is larger than the load average of any other host in the network.

(3) A host in the network fails more than three times today.

(4) The network is partitioned.

Detection of all these events requires special algorithms; therefore, we classify these events as *special events*. A special event can be detected and registered only if the event manager has an algorithm to detect that event. We do not discuss special events in this paper.

4.2. Request Formation

A request consists of two major parts: an *event* and an *action*. When the event is detected, the corresponding action should be fired. An event can be represented as described in the previous section. As for the action, we propose three choices for the clients:

(1) *Notification by Mail:*

This is typically used by a user client. The NEM constructs a mail and send it to the user when an event happens.

(2) *Notification by Inter-Process Messages:*

This is used by a client process; the NEM sends a message to the client process when an event happens.

(3) *Activating a Process:*

Both user clients and process clients can specify a command process to execute on behalf of the client when an event happens.

In some cases, if the action of a registered event is not fired within certain period after the event happens, then the action should not be fired at all. For example, if the event *"the workstation jupiter is lightly loaded"*, is detected at 12:00, then the action *"start a remote typesetting*

process on jupiter" should not be fired after, say, 12:05, because the reported information has lost its validity. Therefore, we can attach a *validity interval* with a request so that the event manager will not fire the action if the time constraint can not be kept.

Some request may specify an event that is not likely to happen. After a long period, requests like this may accumulate in the request table and affect the performance of the event manager. A deadline can be specified so that a request can automatically get deleted after the deadline.

Based on the above discussion, we can use the following grammar to represent a request:

```
request: <basic_request>
       | <within_request>
       | <deadline_request>
       ;
basic_request : <event> <action> ;
within_request: <basic_request> WITHIN <validity_interval> ;
deadline_request: <basic_request> <deadline>
          | <with_in> request <deadline>
          ;
action  : MAIL
     | MESSAGE
     | <command>
     ;
```

Existential quantifiers can be introduced to pass some parameters from a detected event to its associated action. The general form of the basic request becomes:

If there exist $x_1, x_2, ..., x_n$, such that $event(x_1, x_2, ..., x_n)$ occurs,
then $action(x_1, x_2, ..., x_n)$ should be fired.

Here is an example request:

If any host x has a load average larger than 10.0,
then kill all background processes on host x within 5 minutes.

We describe how a Lisp pattern matcher can be used to handle such a request in Section 5.

4.3. Event Matching

The major task of the network event manager is the event matching. Figure 5 shows all the data objects used in the event matching task — the status database, the snapshot, the request table, , the clock, and the arriving requests. The status database is maintained by the status maintainer. Whenever a status report is available, the current database is updated and the replaced attribute values pushed to the history database. All the event matching tasks are performed on a snapshot copy, which is copied over from the status database to the event manager at fixed intervals. The snapshot copy is introduced to avoid the synchronization overhead involved in simultaneous reading and updating the status database.

Figure 6 shows an outline of the main routine of the event manager. There are three cases where an event matching can occur. When a request arrives, it is first matched against the snapshot; if there is a match, the action specified in the request is fired immediately; otherwise, the request is entered to the request table.

Whenever the snapshot is updated, all the events registered in the request table will be matched against the snapshot to check if some event happens. This is called *time-triggered matching*. The delay between event detection and action firing is affected by the interval of time-triggered matching.

Whenever there is an update in the status database, the status maintainer can broadcast this information, and the event manager can then copy the new status database to the snapshot

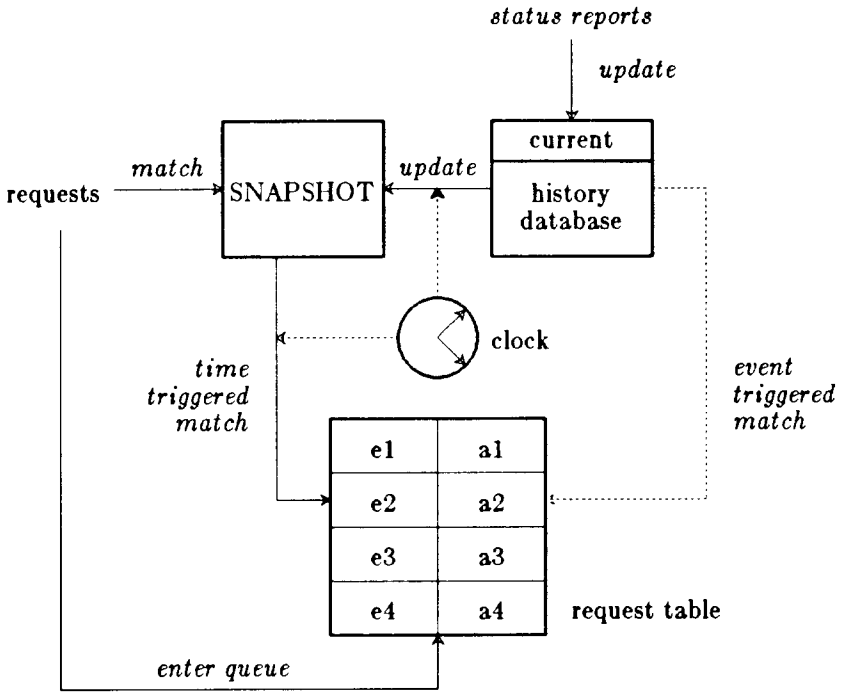


Figure 5. Data objects used in event matching

and see if the new snapshot now matches any registered event in the request table. This is called *event-triggered* matching. In general, this approach is more expensive, since the status database can be frequently updated. However, if some attribute values are critical for the application environment, this approach has the advantage of reducing the delay of action firing. An implementation can incorporate either matching approach, or both.

Note that the size of the request table must be controlled; otherwise, the event matching task cannot be completed during the timer interrupt interval. There are several approaches to solving this problem. One approach is to give requests that are attached with shorter validity intervals higher priorities in event matching. Another approach is to enforce a deadline for each request (see Section 4.2) and delete the request when the clock reaches the deadline. The final approach is to introduce more event managers when the table reaches certain limit. This problem is discussed in Section 7.

5. IMPLEMENTATION

To experiment with our ideas, we have implemented prototypes of Local Event Manager[14] and Network Event Manager[15] on a set of VAX780s at Berkeley using the networking facilities available in Unix 4.3 BSD[16]. Both systems consider only events without range or time specifications. Also, both of them implement only time-triggered matching. This section describes our experience with these two prototypes.

5.1. The Prototype Local Event Manager

This event manager is implemented in a local environment. Only users on the same machine can post requests. The event manager views the network as a collection of host objects. Only the current database of the host attribute values are used in event matching. Therefore, only state events and assertion events can be detected. The current database is stored by system daemons periodically in /usr/spool/rwho; see the Unix command *rwho(1C)* for details.

The request is represented using the Lisp syntax:

```
{ Main routine of the event manager }
const INTERVAL=60;
<other declarations>
{ Set up the interrupt timer; TimeMatch is the interrupt-service routine }
SetupTimer(TimeMatch, INTERVAL);
while(true)
      msg:=read_msg();
      case msg.type of
         { match a new request }
         REQ_ARRIVAL: result:=single_match(msg.request);
                  if (result=NO_MATCH) add_request(request);;
         { event-triggered matching; match all requests when new status is received (optional) }
         {TimeMatch is called to update the snapshot before matching}
         {Note that current database is updated by the status maintainer, not by this layer}
         NEW_STATUS:  TimeMatch();;
      endcase
endwhile

{ This is declared as a monitor to avoid synchronization problems on the status database }
monitor event_match =
var status_table: set of status_report
<other declarations>
begin

   {This function matches a single request}
   entry function single_match(r: request);
   begin
      { within_request and deadline_request not considered here }
      if match(r.event) then
      begin
         r.action();
         remove_request(r);
         return true;
      end
      else
         return NO_MATCH;
   end

   {This function is called when a timer interrupt occurs;
   it copies the status database to the snapshot, and does a time-triggered match}
   entry function TimeMatch();
   begin
      UpdateSnapShot();
      table_match();  { time-triggered match }
   end
end

{ This is declared as a monitor to avoid synchronization problems on the request table }
monitor request_management =
var request_table: set of request;
<other declarations>
begin
   {This function matches all the requests when new snapshot is obtained}
   entry procedure table_match();
   begin
      for r in request_table do single_match(r);
   end

   entry function add_request(r: request);
   entry function remove_request(r: request);
end
```

Figure 6. Outline of the event manager

(**if** *event* **then** *actions*)

For example, a request can be†:

(if ((equal host ucbbuddy) (lessp load 3) (lessp users 15))
then (echo ucbbuddy is lightly loaded) (mail John < message))

This requests that the message "ucbbuddy is lightly loaded" be displayed on the terminal, and a message sent to john when the load average on the machine ucbbuddy is less than 3 and the number of users on the machine is less than 15. The message sent to John must be prepared by the user himself.

Datagrams in *Unix domain* [16] are used for inter-process communication. The socket number of the event manager is stored in a well-known file. A client interface reads this file to find out the address of the event manager. The client interface is provided for checking the request syntax and sending legal requests to the event manager. It also forks a process that waits for event notification from the event manager and then takes the requested actions when the event notification arrives.

The event manager is implemented as two communicating processes. The first process, implemented in Franz Lisp[17], performs the actual event matching task. The second process, implemented in C, accepts requests from the client interface, obtains the network status information, and sends the received requests and the status updates to the Lisp process.

The Lisp process is able to handle requests like

if ((assign host host1) (equal status up))
then (echo (using host host1) is up)

This is a request for printing out the names of all the hosts which are up on the terminal. Note that host1 is a *pattern* variable that gets assigned to the names of hosts which are up and is substituted in the action part of the request. The substituted actions are then sent to the C process for execution. This is a way to implement the existential quantifiers we mentioned in Section 4.2.

5.2. The Prototype Network Event Manager

Datagrams in the *Internet domain*[16] are used for inter-process communication in this implementation. Because the implementation is done in the *Internet domain*, users on other hosts can also post requests at the event manager. But, unlike the local event manager, the socket number could not be stored in a file as files are not shared between hosts on the Berkeley Vax network. This socket number was, therefore, made known explicitly in advance to the client interface on each host.

This event manager views the network as a collection of hosts and users. The status information is also obtained periodically from /usr/spool/rwho. However, each time before the current snapshot is updated to a new value, it is saved as the previous snapshot. Figure 7 shows the relationship between the two snapshots and the status database. These two snapshots are necessary for detecting differential events.

The event matcher is implemented as a single C process. Event and request representation follows a subset of the grammar described in Section 4.1 and 4.2. The request can be sent through a C program interface routine or by interactive command input. A simple natural language user interface[18] is also provided so that a user can post requests through "well-structured" English sentences. For example, a request can be typed in directly by a user in the following form:

† The format shown here is slightly different from the one in Section 4.1.

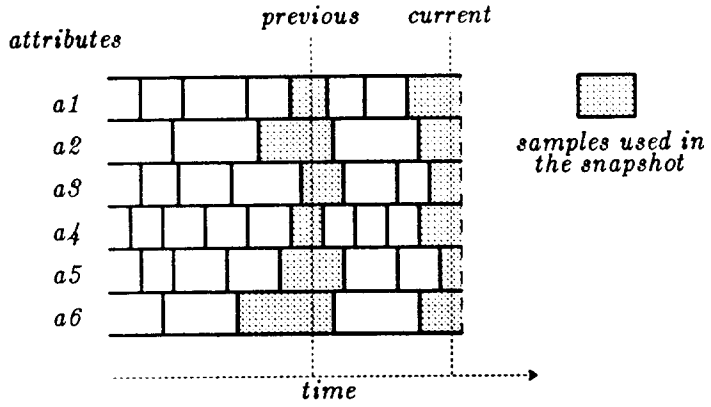


Figure 7. The current snapshot and the previous snapshot

If the load average on ucbarpa is greater than 3.5,
then send me a mail.

when the specified event is detected, the event manager constructs the following message from the request, and sends it to the user‡:

The load average on ucbmonet is now greater than 3.5. It is currently 3.81.
This event was detected at 10:12am, Jul. 3, 1986.

- Event Manager

5.3. Discussion

The two implementations have their own advantages and disadvantages. The Lisp & C implementation handles the logical combination of events in a neat way using a pattern matcher. Further, it is easier to implement inferencing from a series of events in Lisp. However, it suffers from the problem that global data must be transmitted back and forth between the C process and the Lisp process.

Both implementations suffer from the limitation that a request once registered cannot be withdrawn until it is executed. Therefore, if a user registers a request (without deadline) that never gets satisfied, it will remain stored forever with the event manager. One option is to return a stub to the client for a request registered. The client can use the stub to ask Event Manager to delete or renew the request. Another option is to provide a command that lists all the requests (with request IDs) registered by a user in the event manager. In this case, a user does not have to keep stubs around. This is the approach used by the Unix *at(1)* command.

Our experience with the C Network Event Manager showed that a shared file system could have facilitated our implementation. In that case, we could have stored the socket number of the NEM in a shared file, accessible to all the client interfaces. Since the client interface does not have to remember the socket number of the NEM, the NEM can migrate from host to host much more easily. Furthermore, locating newly created NEMs would not be a problem either; this information can be stored in a file. Note that we are then basically using shared files as a simple alternative to the name server. For a discussion of the name server versus the network file system as an object locating mechanism in computer networks, see[19]. The SUN clusters in our department use a Network File System[20]. We are therefore planning an implementation of the NEM on the SUN clusters.

‡ This feature is implemented by Milton G. Howard.

6. APPLICATIONS

In this section, we consider three applications of the network event manager: supporting pulsating computations, acting as a user watchdog, and providing an active bulletin.

6.1. Pulsating Computations

One of the goals of the Network Event Manager (NEM) is to support distributed computations by handling status changes in a network. To have a concrete discussion on this subject, we assume that the environment is a local area network with computation nodes sharing a network file system. A typical example is a Sun workstation cluster. We first define a class of distributed computations that can best exploit the nature of this environment and then examine how the NEM supports the dynamic restructuring of this class of computations.

A *Pulsating Computation*† goes through three phases consecutively:

(1) *Expansion Phase:* At this phase, a *master* process decides the number of slaves it needs for the computation, selects slaves from the network nodes, and then initiates computation tasks on the slaves.

(2) *Processing Phase:* At this phase, the computation tasks are performed on the slaves.

(3) *Consolidation Phase:* At this phase, the master collects the results of the tasks and performs some coordination tasks if necessary.

A pulsating computation can have a single pulse or multiple pulses (Figure 8). Each *pulse* consists of the above three phases.

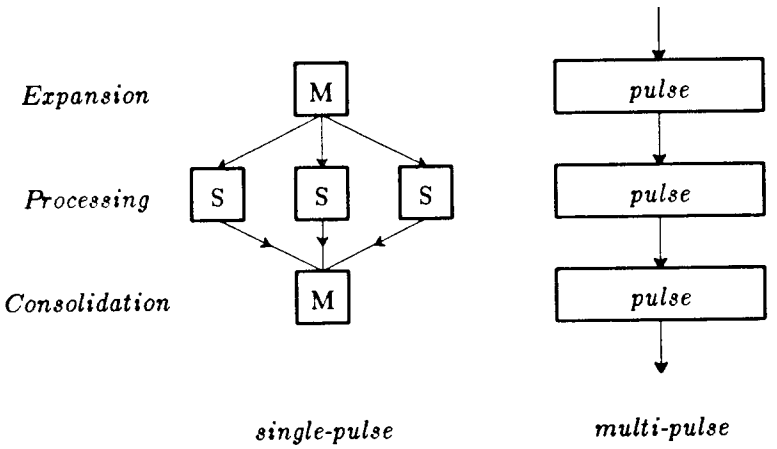


Figure 8. Pulsating Computations

Examples of single-pulse computations are:

(1) Compilation: program files can be compiled separately by slaves. After the compilation tasks are completed, a linking operation can be performed by the master at the consolidation phase to produce the final object code.

(2) Typesetting: separate chapters can be processed separately by slaves. At the consolidation phase, these processed chapters are combined together by assigning the page numbers, collecting the references, and renumbering the figures and tables.

Examples of multi-pulse computations are:

(1) Clock synchronization: at each pulse, a master time daemon sends its clock time to the slave time daemons on all the hosts; each slave updates its clock, calculates the

† Luis Felipe Cabrera suggested the idea of pulsating computations to us; we have defined it using slightly different terminology here.

difference, then sends the results back to the mater time daemon. Based on the collected results, the master time daemon calculates a global time, and updates its own clock. The algorithm used in TEMPO[4], a clock synchronization system, is similar to what we have described here.

(2) Computer animation: During expansion phase, a master node selects a number of slave nodes to compute successive frames of a cartoon. During processing phase, the selected slaves compute the assigned picture frame. At the consolidation phase, the master collects the results for display and decides when to initiate the next pulse based on the timing and buffering constraints. This is similar to one of the applications of the Worm program[3].

Pulsating computation achieves parallelism by explicitly creating processes on local and remote nodes. If all processes access data from the shared file system, then there is no need for data transfer (except for some control messages), between the master and slaves†. It is desirable for the following two conditions to hold in partitioning the tasks among slaves so that the synchronization overhead can be minimized:

PC1. $R(i) \cap W(j) = \phi$, for all i and j
PC2. $W(i) \cap W(j) = \phi$, for all i and j

where R(i) denotes the read set of slave i
and W(j) denotes the write set of slave j.

Note that all the four aforementioned examples of pulsating computations satisfy PC1 and PC2. Many computations can also be restructured to follow PC1 and PC2 for increased parallelism. The restructuring involves pushing tasks that are inherently serial into the consolidation phase. The Unix C compiler, *cc(1)* command, is a good example; program files can be independently compiled, followed by resolution of cross references in the linking phase.

Assuming that the communication subsystem delivers messages reliably and a reliable file system is available, several events may still affect a pulsating computation:

E1. A slave node crashes. It is necessary to start a new slave process on a different node to replace the old one so that the pulsating computation can continue.

E2. A slave node becomes overloaded for certain period. It may be necessary to migrate the slave process to another node to avoid further delay of the processing phase.

E3. The master node fails. A new master should be selected, which can collect computed results from slaves and start new pulses.

Note that E1 and E2 can be handled only if the following condition holds:

PC3. The slave processes are restartable,
i.e., initiating a new copy of a slave process does not lead to any inconsistencies.

For many applications, including those we discuss in this paper, we have found that PC3 holds. In fact, it turns out to be pretty hard in distributed systems to ensure that some action is taken *exactly* once, i.e. not zero times and not more than once[21]. We believe therefore that PC3 is a reasonable assumption for many applications. However, the assumption we made here is that if a node crashes, it can be detected, and all processes on the crashed node will cease operation; otherwise, consistency problems will arise if a supposedly dead slave is still accessing the data files accessed by a new slave.

There are several approaches for detecting events E1 through E3:

(1) The master node can periodically query each slaves for its status. This is the approach used in most remote procedure call (RPC) mechanisms to guarantee that the callee is still alive.

† Parallelism may be degraded if all the slave processes have input/output- intensive tasks and the shared network file server becomes the bottleneck.

However, if the master fails, the pulsating computation can not proceed to the next pulse. Furthermore, subsequent slave failures will not be detected. Human intervention would be necessary to determine the status of the computation and to restart a new master or to abort the computation.

(2) The slave nodes can detect the status of the master, and run an election algorithm when the master fails. This is the solution used in TEMPO[5]. For many applications, however, this approach may incur too much overhead.

(3) The third approach, which is based on a reliable network event manager (see Section 7) can be outlined as follows: Before the master starts any pulse, it creates a *guardian* at a node other than the resident node of the master. All messages communicated between the master and the slaves or the event manager must also be sent to the guardian. Normally, these messages (mostly task initiation and completion messages, registering requests, etc.) should not introduce a large overhead since many pulsating computations use the shared file system for data access. The master then registers the following requests at the event manager:

R1: event: failure of the master node

action: notify the guardian and delete requests R2, R3, and R4

R2: event: failure of the guardian

action: notify the master (start a new guardian)

R3: event: failure of any slave node

action: notify the master (start a new slave)

R4: event: overloading of any slave node

action: notify the master (kill the old slave, start a new slave)

After these requests are registered, the master forks processes on remote nodes and enters an event loop. It waits for either a notification from the event manager, or for the task completion or error messages from the slave processes. In general, a master executes the following in each pulse:

```
register requests at NEM;
fork slave processes on remote nodes;
while (not all slaves complete their tasks)
   message = read_message();
   case message.type of
      event_notification: handle events;
      task_completion: add the slave name to the task completion list
      task_error: handle errors;
   endcase
endwhile
perform consolidation tasks;
decide if and when the next pulse should be initiated.
```

If the master fails, the guardian will examine the state of the master and take over the pulsating computation. We will not go into the details of the recovery routines used in the pulsating computation. However, there are three possibilities for removing the guardian we introduced here:

(1) If a pulsating computation has to be aborted anyway (due to the nature of the application) whenever the resident node of the master crashes, then there is no reason to create the guardian.

(2) For multi-pulse pulsating computations, if the master fails, the event manager can abort all the slaves and create a new master, which should start execution from the end of the previous pulse. Using this scheme, we can remove the guardian, but the master must checkpoint its state and save it in the shared file system whenever it finishes a pulse.

(3) The final possibility is to store all the messages to or from the master to the shared file system. However, this scheme can be expensive because each message will incur a file update.

Since many computations can be transformed to the form of pulsating computations, we believe that it is more sensible to make the event manager reliable than to construct fault tolerance routines for each pulsating computation. The scheme we propose here can not handle all possible failures that may affect a pulsating computation, but it takes care of most common failures in an efficient and practical way.

6.2. User Watchdog

Users in a network environment often need to detect events like the following for various reasons:

(1) John logs on to *ucbkim* : Need to contact John as soon as he logs on.

(2) *Ucbernie* comes back: Need to get some data stored on *ucbernie*.

(3) The load average on *ucbarpa* is below certain threshold: Need to run a large job on *ucbarpa*.

(4) The only user on the workstation *nutmeg* just logs out: Need to grab that workstation to draw some VLSI layouts.

A user can waste his precious time in periodically detecting these events, or he can write shell scripts to detect these events periodically†, or the network event manager (NEM) can watch for these events and notify the user when events of interest to the user take place.

A user does not even have to be there when the events of interest happen; the NEM can send him a mail, and the user can later read the mail to find out when that event took place. Or, if the user wishes to execute a certain command on the occurrence of an event, the NEM can directly execute the command specified by the user. The Unix *at(1)* command is similar to NEM in this sense but different from NEM because

(1) The *at* command is concerned only with *timing events*. It accepts requests that specify a time and a command to be executed at that time.

(2) The *at* command handles requests only for a local machine.

The NEM can be used to implement a distributed *at*. The clocks on various nodes can be incorporated in the network abstraction, and an event can be defined as one of the clocks reaching a particular value. If a notion of global clock is available (through clock synchronization), then it can be used as the time reference for all requests.

6.3. Active Bulletin Board

An event manager can also act as an agent that collects event information from various processes and distributes the information to interested clients. This is like a bulletin board, where people post events for those who might be interested; the only difference here is that our bulletin board knows who is interested in what information and delivers the event information immediately. When used as an active bulletin board, an event is not necessarily detected from the changes in the network status; it may be supplied directly by some process or user.

For example, in a network that consists of several relocatable file servers, clients are unable to access a crashed server's files until it is rebooted; each file access may result in a broadcast looking for that crashed file server (because the file servers are relocatable), followed by a timeout[19]. The event manager can help eliminate these broadcast messages by acting as an agent between the clients and the file servers. Both the clients and file servers do not have to know each other. When a client node discovers that a particular file server has crashed, it posts its request with NEM, which specifies its interest in knowing about the recovery of the file server. The client node can then stop trying to access any files stored at the crashed file server. When a

† For example, see the *watchfor* command in [22].

file server comes back, it declares its recovery to the event manager. The event manager then sends notification messages to all client nodes who are interested in this event. In this scheme, the event manager has to include the relocatable file servers in its view of the network objects. A naming scheme should also be provided so that a client can name a file server easily.

Another example comes from the software development process. A software development system consists of many objects, e.g., requirement specifications, design documents, source modules, test cases, maintenance records, and software personnel. Examples of events that may affect the software development process are:

(1) A change in the requirement specification.

(2) Restructuring of certain software modules.

(3) Modifications of certain procedure interfaces.

Detection of these events may be critical for the coordination of the software development effort. However, with a large software development team, distributed on several machines, it is very difficult to notify all the people who may be interested in an event. To solve this problem, we can build a special event manager that does this coordination task. This event manager has an abstract view of all important software objects. When a software personnel finishes the modification of some software object, he posts this event at the event manager. If a person is interested in some of the modifications, he posts a request at the event manager. Upon a matching of an event and a request, the event manager notifies the requesters. Using this approach, the requester does not have to check the events himself and the event-poster does not have to know who are interested in the event he posted. Ramamoorthy, Garg, and Aggarwal proposed a rule-based *activity manager*[23] for solving the problem we described here in a single-machine environment using a somewhat different approach. The activity manager restricts the programmers' activities according to some rules, while the event manager simply accepts requests and event reports, and performs the matching task; rules are not enforced.

Based on the above examples, we feel that the idea of event manager can be applied to any system that has the following properties:

(1) The system can be abstracted as a set of objects that can change their attribute values synchronously or asynchronously.

(2) A set of clients are interested in the change of these attribute values, but they do not have enough resources to detect these events in real time.

(3) Timing problem is not critical in the sense that consistency does not depend on ordering and time of detection of events.

7. FUTURE WORK

We still have to extend our prototype implementations to handle more powerful types of events (e.g. events with time information). We have found prototype event managers useful to monitor the status of the network for us, for instance to keep a watch on a remote crashed host and to inform us when it comes up. But, work is required in the direction of building applications, such as pulsating computations, that utilize the services of prototype event managers. We have not addressed several important issues regarding event managers, for example, the timing problem caused by variable delays in the communication subsystem, construction of multiple event managers, and reliable event managers. We briefly discuss how we might tackle them in future.

Timing Problem

In distributed systems, it is impossible to perfectly synchronize clocks and ensure message deliveries with zero delay. Also, sampling of network status can only be done at discrete intervals of time, causing uncertainities about the status between the sampling periods. All these factors lead to uncertainity about the exact time of occurrence of events. For example, suppose the following information is provided by the status maintainer (t1 and t2 are the clock times at the local host, $t1 < t2$):

S1: The number of users on ucbarpa is 18 at clock time t1
S2: The number of users on ucbarpa is 19 at clock time t2

Assume that the maximum difference between any two clocks is δ and the sampling rate is large enough for recording the changes of the number of users. From S1 and S2, we can deduce the following events:

E1: The number of users on ucbarpa increased between t1+δ and t2-δ — a differential event.
E2: The number of users on ucbarpa is larger than 18 between t1+δ and t2-δ — an assertion event.

The timing problem becomes more involved for the complex event. Suppose that we want to detect a network partition event as shown in Figure 9, and the following information is inferred by the event manager:

E1: Link L1 failed between t1-δ1 and t1+δ2.
E2: Link L2 failed between t2-δ3 and t2+δ4.
E3: Link L1 recovered between t3-δ5 and t3+δ6.

From the above, we can infer that:

E4: Link L1 was in the failure state between t1+δ2 and t3-δ5.
E5: Link L1 AND Link L2 were in the failure state between t2+δ4 and t3-δ5
E6: Link A OR Link B was in the failure state between t1+δ2 and t3-δ5

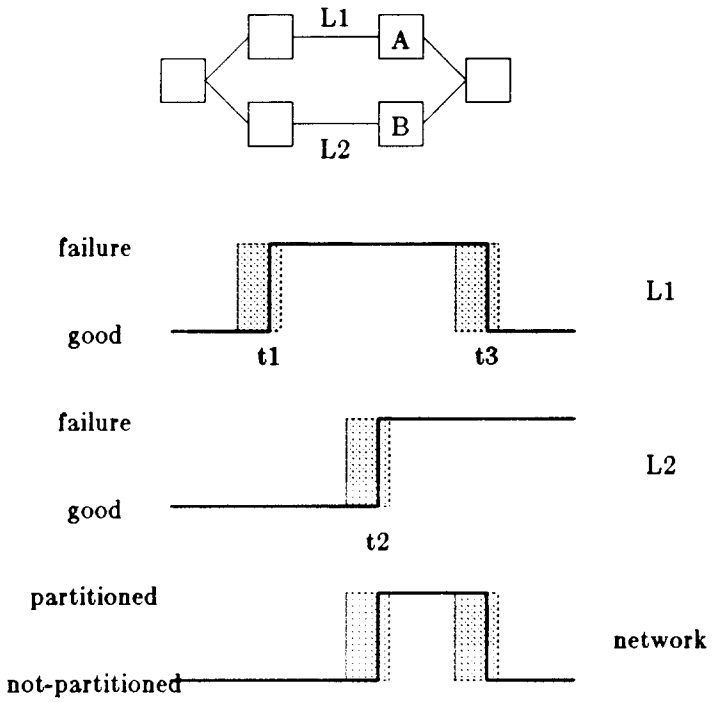


Figure 9. Detection of a network partition

Note that the AND, OR, and NOT operators we described in Section 4.1 are associative and commutative for the purpose of combining timing estimates.

If sampling rate is very slow, or δ is large, it becomes almost meaningless to associate timing information with requests. Inconsistencies may also arise if timing information is used for database updates or making assertions about global state of the network (in the partition example, if δ is large, it is possible that no partition existed at any instant). This complicates application of Event Manager to transaction-oriented applications. Despite these limitations, we believe that

Event Manager is a useful service for the users as well as distributed computations that are not transaction-oriented (a large class of applications can do without the overhead of distributed transactions). For the kind of applications for Event Manager we have given in this paper, timing information is not critical. One reason is that network status does not change too rapidly; if a node crashes, it usually remains crashed for a long enough period that clock and communication errors do not become a factor.

Multiple Event Managers

If a single event manager can not handle all the requests in time, many requested actions will be delayed and the service will be degraded. Multiple event managers can potentially share the responsibilities and improve the performance. The number of event managers also can be dynamically decided by the event managers themselves based on the demand. The problems here are (1) how to divide up the requests from users among the event managers and (2) how to order the firing of actions. If event $r1.e$ happens before event $r2.e$ according to the timing estimates, then presumably action $r1.a$ should be fired before $r2.a$. This requirement can be achieved on a single event manager by delaying the execution of certain actions, but it is difficult to achieve when actions can be fired independently from different places.

Reliable Event Managers

A reliable event manager becomes necessary if applications are to trust the event manager for reliable detection of events. A *guardian*† process can be located on a node other than the resident node of the event manager. Essentially, the state at the event manager is the request table only. Therefore, the goal is to keep the guardian posted of all the updates made to the request table. When the event manager fails, the guardian must reconstruct the request table, notify each client interface of its existence, and create a new guardian.

8. CONCLUSION

The Network Event Manager provides an integrated network service for the detection and handling of asynchronous events in computer networks. It accepts requests from network clients, and coordinates the detection and notification of events. This network service will become increasingly important in the future as a basic support for distributed pulsating computations, which explore the parallelism of the local area networks. The general idea of event management can also be applied on other systems as a way to structure the dynamic information and to make it available to clients.

Acknowledgement

We would like to thank Hock San Lee and Milton G. Howard for their contributions to the two prototype implementations. We would also like to thank Luis Felipe Cabrera for suggesting to us the idea of pulsating computations, and thank our colleagues Vijay Garg, Anupam Bhide, and Jaideep Srivastava for valuable comments which have helped in writing and improving this report.

References

1. Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

† This guardian is different from the guardian we mentioned in Section 6.

2. *The SunView System Programmer's Guide,* Sun Microsystems, Inc., Mountain View, California, October 1985.

3. J.F. Shoch and J.A. Hupp, "The "Worm" Programs - Early Experience with a Distributed Computation," *CACM,* vol. 25, no. 3, pp. 172-180, March 1982.

4. Ricaardo Gusella and Stefano Zatti, "TEMPO: Time Services for the Berkeley Local Network," *UCB/CSD 83/163,* Universtity of California, Berkeley, December 1983.

5. Riccardo Gusella and Stefano Zatti, "An Election Algorithm for a Distributed Clock Synchronization Program," *UCB/CSD 86/275,* University of California, Berkeley, December 1985.

6. Danny Dolev and Joe Halpern, "On the Possibility and Impossibility of Achieving Clock Synchronization," *Proceedings of the Sixteenth Annual ACM STOC,* pp. 504-511, 1984.

7. Leslie Lamport and P. M. Melliar-Smith, "Byzantine Clock Synchronization," *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing,* pp. 68-74, 1984.

8. Joseph Y. Halpern, Barbars Simons, and Ray Strong, "Fault-Tolerant Clock Synchronization," *Proceedings of the Third ACM Symposium on Principles of Distributed Computing,* 1984.

9. Anupam Bhide and C. V. Ramamoorthy, *Real-Time Fault-Tolerant Clock Synchronization,* Computer Science Division (EECS), University of California, Berkeley, June 1986.

10. Ramon Cacéres, "Process Control in a Distributed Berkeley Unix Environment," *UCB/CSD 84/211,* University of California, Berkeley, December 1984.

11. Luis Felipe Cabrera, Stuart Sechrest, and Ramon Cacéres, "The Administration of Distributed Computations in a Network Environment," *UCB/CSD 86/268,* University of California, Berkeley, November 1985.

12. K. Main Chandy and Leslie Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. on Computer Systems,* vol. 3, pp. 63-75, February 1985.

13. Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler," *Unix Programmer's Manual,* vol. 2, 1984.

14. Hock San Lee, "A Rule-Based Event Manager," *CS 298 Final Report,* Computer Science Division (EECS), University of California, Berkeley, December 1985.

15. Yih-Farn Chen, "A Prototype Implementation of the Event Manager," *CS 292 Final Report,* Computer Science Division, University of California, Berkeley, December 1985.

16. Samuel J. Leffler, William N. Joy, and Robert S. Fabry, *4.2 BSD Networking Implementation Notes,* CSRG, Computer Science Division (EECS), University of California, Berkeley, July 1983.

17. John K. Foderaro, Keith L. Sklower, and Kevin Layer, "The FRANZ LISP Manual," *Unix Programmer's Manual,* vol. 2, University of California, Berkeley, June 1983.

18. Yih-Farn Chen, "The Front End Interface for the Event Manager," *CS 288 Final Report,* Computer Science Division, University of California, Berkeley, December 1985.

19. Brent Welsh and John Ousterhout, "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System," *UCB/CSD 86/261,* University of California, Berkeley, October 1985.

20. *Sun's Network File System,* SUN Microsystems, Inc., Mountain View, California, 1985.

21. B. W. Lampson, M. Paul, and H. J. Siegert, *Distributed Systems - Architecture and Implementation,* pp. 365-370, Springer-Verlag, 1981.

22. Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment,* p. 145, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

23. C.V. Ramamoorthy, V. Garg, and R. Aggarwal, "Environment Modeling and Activity Management in GENESIS," *SoftFairII: Second Conference on Software Development Tools,*

Techniques and Alternatives, December 1985.