Monitoring Distributed Systems

A functional model of monitoring in terms of the generation, processing, dissemination, and presentation of information can help determine the facilities needed to design and construct distributed systems.

Masoud Mansouri-Samani and Morris Sloman

Monitoring, the dynamic collection, interpretation, and presentation of information about objects or software processes, is needed for the management of distributed systems or communications networks, as well as for debugging, testing, program visualization, and animation [1]. The information gathered is used to make management decisions and perform the appropriate control actions on the system (Fig. 1). In this article we discuss only the monitoring of object-based distributed systems, in particular the use of monitoring as a management tool. (We leave the discussion of control actions for another forum: unlike the passive monitoring process, control actions change the behavior managed systems, and therefore should be addressed separately.)

There are a number of fundamental problems associated with monitoring of distributed systems. Delays in transferring information from the place it is generated to the place it is used means that it is out of date. Thus it is very difficult to obtain a global, consistent view of all components in a distributed system. Because variable delays in the reporting of events may result in the sequence of those events being recorded in the incorrect order, some form of clock synchronization is necessary to provide a means of determining causal ordering. The number of objects generating monitoring information in a large system can easily swamp managers, thus necessitating the filtering and processing of information. Another problem is that the monitoring system may itself compete for resources with the system being observed and so modify its behavior.

To overcome these problems, a monitoring system must provide a set of general functions for generating, processing, disseminating, and presenting monitoring information.

MASOUD MANSOURI-SAMANI is a research student in the department of computing at Imperial College.

MORRIS SLOMAN, Ph.D. is a reader in the department of computing at Imperial College.

Concepts and Terminology

A managed object is defined as any hardware or software component whose behavior can be controlled by a management system. (In this article, we will refer to managed objects as objects, unless otherwise noted.) The object encapsulates its behavior behind an interface, which hides the internal details that are vital for monitoring purposes. For this reason, the concept of encapsulation in object-based distributed systems causes a problem as far as monitoring is concerned. The interface of a managed object can be divided into two parts [2] (Fig. 2):

- An operational interface that supports the normal information-processing operations, fulfilling the main purpose of the service provided by the object.
- A management interface that supports monitoring and control interactions with the management system.

The International Standards Organization (ISO) has defined a series of standards for the management of the communication system for Open Systems Interconnection (OSI). These define managed objects as a representation of a managed resource; however, managed objects can also be defined as part of the monitoring system rather than part of the managed system [3]. The OSI concept of an object is a representation of a resource that is being managed, so it corresponds to the management interface shown in Fig. 2.

Monitoring can be performed on an object or a group of related objects (a monitoring domain). The behavior of an object can be defined and observed in terms of its status and events. The status of an object is a measure of its behavior at a discrete point in time and is represented by the current values of a set of status variables (attributes) contained within a status vector [4]. An event is an atomic entity that reflects a change in the status of an object. The status of an object has a duration in time (such as "process is idle" or "process running"); an event occurs instantaneously ("message sent," "process started," or "counter reaches a threshold"). Three kinds of events are of interest in distributed systems [5].

A control-flow event represents a control activity and is associated with a control thread. Such an event occurs when a process or the operating system reaches a previously defined statement. For example:

- Process P1 enters/leaves procedure Fred for the *n*th time.
- The operating system enters the scheduler.

A data-flow event occurs when a status variable is changed or accessed. For example:

- Variable *a* of process P1 is assigned value X.
- Variable *b* in the third invocation of procedure Fred is read by process P2.

Process-level events show the creation and deletion of processes and the interactions and data flow between them. For example:

- Process P1 starts.
- Process P1 sends message m to process P2.
- The number of waiting processes in queue w is incremented by one.

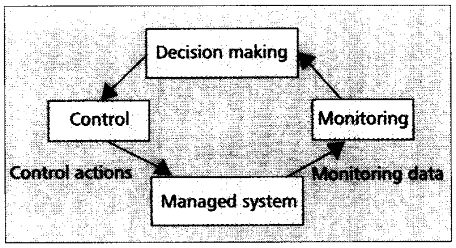
Control-flow and data-flow events are internal events that are related to the local state of a component or object and are not visible outside it, unless explicitly made visible at the management or debugging interface. Such events are particularly useful for debugging purposes. Note that debug tools often create a new "interface" to make visible internal events and states that are not normally visible at either the operational or management interface to an object. Internal events thus violate the encapsulation of objects and are more appropriate for analyzing the behavior of a single component.

Process-level events can be considered as external events that represent the external behavior of an object and its interactions with other objects. These events are of particular interest in management. A generalized monitoring system should allow observation of a combination of these events. Many monitoring tools enable the users to specify and detect only process-level events such as interprocess communication, because these events do not violate the encapsulation of objects and are at the correct level of abstraction for analyzing the behavior of distributed systems [1, 6, 7]. OSI management events are generated whenever a managed object is created/deleted, an operational/administrative state change occurs, or changes occur in nonstate attributes such as names or important operational parameters [8].

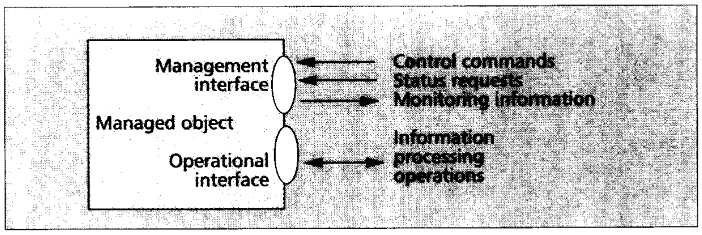
Events can also be classified according to their level of abstraction into primitive and combined (or correlated) events: a primitive event signifies a simple change in the state of an object; a combined event is a combination or grouping of other primitive and combined events. This classification is useful for describing the global behavior of a group of objects in terms of the local behavior of every object in the group. Various languages are used for specifying combined events and states. (We describe this in more detail later in this article.)

Monitoring information describes the status and events associated with an object or a group of objects under scrutiny. Such information can be represented by individual status and event reports or a sequence of such reports in the form of logs or histories, as described later.

Time-driven monitoring is based on acquiring periodic status information to provide an instantaneous view of the behavior of an object or a group of objects. There is a direct relationship between the sampling rate and the amount of information generated. Event-driven monitoring is based on obtaining information about the occurrence of events of interest, which provide a dynamic view of system activity, as only information about the changes in the system are collected. Most monitoring approaches use event-driven monitoring, but a generalized monitoring system



■ **Figure 1.** *A system management model.*



■ **Figure 2.** *A managed object.*

must provide both of these complementary techniques to suit various monitoring requirements and constraints.

Monitoring Model

Our survey is based on a general functional model that is derived from the Event Management Model [4], with some changes and enhancements. Our model identifies the following four monitoring activities performed in a loosely-coupled, object-based distributed system:

• Generation: important events are detected and event and status reports are generated. These monitoring reports are used to construct monitoring traces, which represent historical views of system activity.

• Processing: a generalized monitoring service provides common processing functionality such as merging of traces, validation, database updating, combination/correlation, and filtering of monitoring information. They convert the raw and low-level monitoring data to the required format and level of detail.

• Dissemination: monitoring reports are disseminated to the appropriate users, managers, or processing agents.

• Presentation: gathered, processed, and formatted information is displayed to users.

Implementation issues relating to the intrusiveness of the monitoring system, which depend on whether monitoring is implemented in hardware or software and how clock synchronization is achieved for event ordering, provide a fifth dimension for comparing monitoring systems (Fig. 3).

Of the many models that have been developed to describe the monitoring process, one approach is to identify a set of layers such as the model proposed in [9]. The four activities we have described may at first appear to be a layered model with generation as the lowest layer and presentation using the services of the lower layers. A generalized monitoring system, however, may need to perform these activities in various places and in different orders to meet specific monitoring requirements. For example, generated information may be

Generation of monitoring information
- Status reporting
- Event detection and reporting
- Trace generation

Processing of monitoring information
- Merging of traces and multiple trace generation
- Validation
- Database updating
- Combination of events
- Filtering
- Analysis

Dissemination of monitoring information
- Registration of subscribers in the dissemination service
- Specification of information selection criteria

Presentation
- Textual displays
- Time-process diagrams
- Animation of events and status
- User control of levels of abstraction
- User control of information placement and time frame for updates
- Multiple simultaneous views
- Visibility of interaction message contents

Implementation issues
- Special-purpose hardware
- Software probes
- Time synchronization for event ordering

■ **Figure 3.** *Elements of a monitoring model.*

directly displayed by an object without processing or dissemination. Events and reports that are distributed to particular managers can be reprocessed to generate new monitoring information or events. Presentation of information may occur at many intermediate stages. For these reasons, we present the monitoring model as a set of activities that can be combined as required in a generic monitoring service.

Generation of Monitoring Information

Monitoring information is generated by object instrumentation: placing in objects software or hardware probes or sensors that detect events or generate status and event reports (Fig. 4). A sequence of such reports is used to generate a monitoring trace. In this section we describe status reporting, event detection and reporting, and trace generation.

Status Reporting

Status reports contain subsets of values from the status vectors and may include other related information such as timestamps and object identities. Such reports represent the status at a specific instance in time and can be generated periodically [10] or on request (requests may be generated on a periodic or random basis). OSI event management permits scheduling of event reports on both a daily and weekly basis [11].

The Clouds operating system uses an on-demand status reporting scheme [12]. Requests are called probes; every object or process has a predefined or user-defined probe procedure that is executed whenever a probe is received. A probe handler sends a message back to the originator of the request, reporting the status condition of the process.

Status reporting criteria define which reporting scheme to use, what the sampling period is, and the contents of each report.

Event Detection and Reporting

The detection of an event may be internal, from within the object: a function that updates the status vector, for instance, may also check the event-detection criteria. Event detection may also be performed by an external agent that receives status reports and detects changes in the state of the object.

Detection of events may be immediate (real-time) or delayed. For example, signals on the internal bus of a node may be monitored, using a hardware monitor, to detect any changes in the status of the node as and when they occur. This method is used in particular for detection of events in time frames of milliseconds. If events are detected by external agents, there will naturally be a delay between the real occurrence of an event and its detection.

Once an event is detected, a report is generated that contains information such as the event identifier, type, priority, time of occurrence, and the state of the object immediately before and after the event. Event reports may also contain values of other application-specific status variables.

Event and status reports may be generated in one or more stages. A preliminary report containing a minimum amount of monitoring information (such as object and event IDs) may be generated by an object. Such a report could then be sent to an object that can generate a more complete report by adding attributes such as time stamps, event types, or text messages.

Obviously, the type and amount of information contained in a monitoring report depends on the requirements of the users or clients. Independent attributes are those primitive attributes that are assigned to all events and status, such as the time stamp, identity of the object, and so on [13]. Dependent attributes are assigned depending on the type of the event or status. A configuration event report representing a "create process" event, for instance, may contain the identity of the created process.

Trace Generation

To describe the dynamic behavior of an object or a group of objects over a period of time, event and status reports are recorded in time order as monitoring traces. Such traces may be used for post-mortem analysis. In many real-time systems the processing or communication overheads in analyzing and interpreting the monitoring reports "on the fly" would be too high, so they are stored as traces for later analysis. Trace generation may also be necessary when the rate at which monitoring information is received and displayed is too quick for an observer to follow.

A complete trace contains all the monitoring reports generated by the system since the beginning of the monitoring session. A segmented trace is a sequence of reports collected during a period of time. A trace may be segmented due to the overflow of a trace buffer, or the deliberate halting of trace generation that results in the loss or absence of reports over a period of time.

Traces can be formed by the objects that generate the monitoring reports, by intermediate monitoring objects when processing monitored information, or by the final user of such information. Trace information may be held in a temporary buffer before being processed or recorded in persistent log or history files. Because there is a limit to the amount of available storage space, the size of a monitoring trace is also limited. Overwriting older records when the maximum storage size is reached may be appropriate if only the most recent behavior of the system is of interest. Alternatively, trace generation may simply be halted until more space is available [3]. Both of these strategies may result in a segmented trace because some of the reports may be discarded. A capacity-threshold event is generated to indicate storage overflow.

A sophisticated tracing scheme might use a logging service, which can be modeled as a managed object [3], for the long-term storage of reports and other information, in a variety of formats, representations, and orderings. Such a scheme may allow the generation of multiple traces or logs representing different logical views of system activity. Special log records containing a log record identifier, logging time, information contained in the report to be logged, and other related data might also be generated. Various implicit or explicit filtering activities can be performed when such records are generated and stored in log files.

A sophisticated trace generation mechanism may allow reports to be stored in occurrence order if an occurrence timestamp is available. This overcomes the problems of ordering according to arrival order, which may lead to incorrect interleavings of monitoring reports and invalid observations because of communication delays.

Access to the trace reports may be on-demand with the issuing of an explicit request to the storage entity or according to pre-determined conditions. On-demand access enables the processing agent to receive the reports when it has the necessary resources to deal with them or when the communication traffic is low. By using a scanning-and-selection service, the user may be able to specify the particular reports required (those generated by a particular object, for example), the trace file from which they may be read, or the number of reports needed.

Pre-determined conditions may be used to trigger the transfer of reports from a trace-buffer to a remote processing agent when the buffer is full or contains *n* elements, the communication load is below a certain threshold, or the processing load is low [14]. These strategies are particularly useful for reducing communication overheads by sending reports in blocks and therefore economizing on channel setup time.

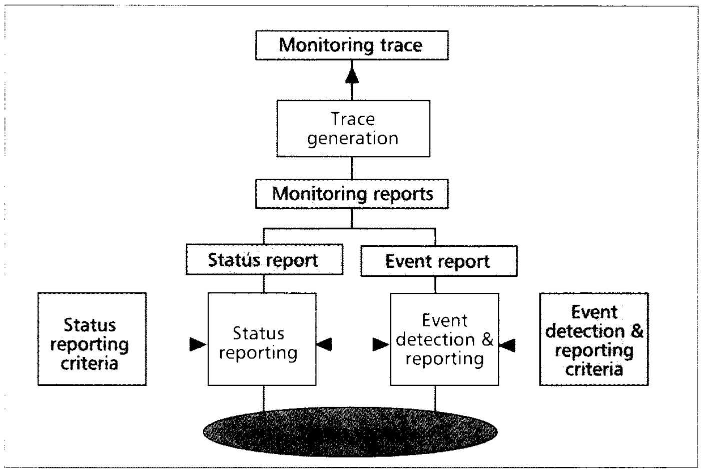
Processing Monitoring Information

After monitoring information is generated, it must be processed. A monitoring service provides various functional units, as building blocks, that can be combined in different ways to suit the monitoring requirements (Fig. 5). Note that these processing functions are often integrated and are performed in different places and at various stages.

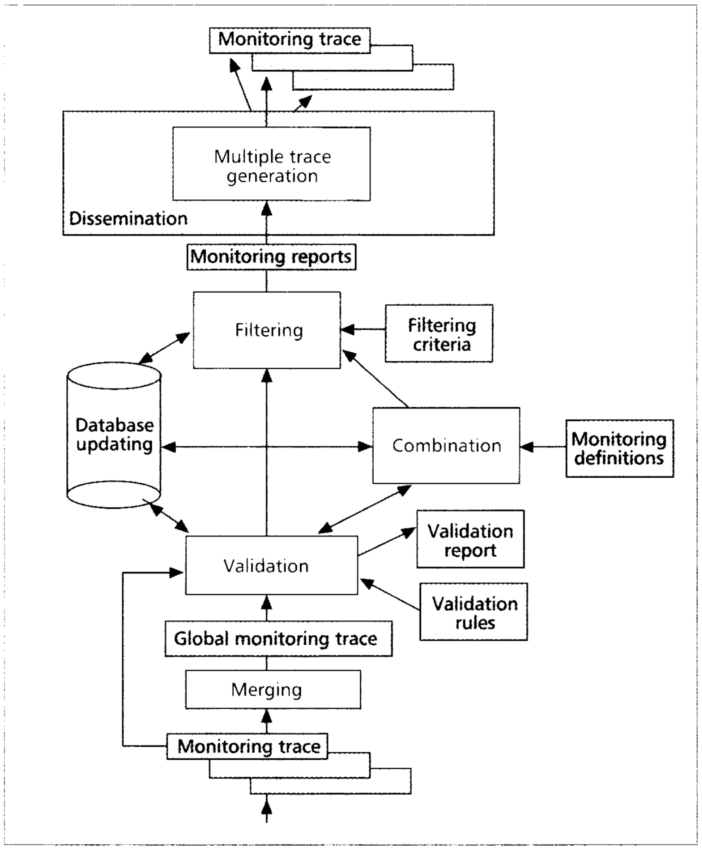
Merging and Multiple Trace Generation

Monitoring traces may be constructed and ordered in various ways to provide different logical views of system activity over a period of time. The selection criteria in determining how monitoring traces are processed include:

• Generation or arrival time stamp, priority, or type



■ **Figure 4.** *Generation of monitoring reports and traces.*



■ **Figure 5.** *Processing monitoring reports.*

elist=>elist2	matches when elist2 occurs immediately after elist1
elist1 > elist2	matches every time elist2 occurs after elist1 has occurred once
elist1 -> elist2	matches when elist2 occurs some time after elist1
elist1 | elist2	matches when either of the event lists occurs
!elist	matches only if elist did not occur
elist PROVIDED condition	matches if condition is true at the time elist occurs

■ **Table 1.** *Examples of State and Event Specification Language (SESL) operators for creating events.*

of report.

- Identity, priority, or type of reporting entity.
- Identity or type of the managed object to which the report refers.
- Identity or type of the destination of the report.

Monitoring traces are constructed according to a trace specification, based on these factors, that specifies how the final traces are formed and what they will contain. Keeping a report from being included in a trace is equivalent to filtering out that report from the point of view of the users who access that trace.

Monitoring traces may be generated from event or status reports as they arrive or from one or more already existing traces, as we will describe.

Merging of Monitoring Traces — A trace segment that contain all the reports related only to one object is called a local trace segment. A set of all the local trace segments from all the objects under scrutiny, over the same period of time, can form a global trace segment.

One of the important activities in a monitoring system is the merging of several monitoring traces into one. Local trace segments from different objects can be merged to generate a global trace segment, representing the global behavior of a set of objects in a particular interval.

Merging may be an iterative operation. A trace generated in this way can be merged with others to generate a more general trace. The original traces may be discarded once merging is complete. Generating one global trace from several local traces is in effect imposing a linear (total) ordering on the event and status occurrences within the system. An ordered trace is simpler to understand and can be easier to work with, but a linear stream may be misleading because it implies an ordering between every pair of event or status reports, even when they are completely unrelated. A partial ordering can more accurately reflect the behavior of a distributed system [15]. A general technique for obtaining the required partial ordering is described in [16]. To achieve this ordering, a vector of logical timestamps is associated with each event. The ordering of events can be determined by comparing the vectors.

Generating Multiple Traces — A monitoring trace can be used to generate several other traces, representing various logical views of object or system activity. Selection of reports from a trace segment may be based on a combination of destination, priority, report type, or other factors. For example, a trace could be generated with all high priority security events, so that they can be processed first, while another trace may be generated with all the accounting events destined for a particular process. A trace generated in this way can be used to generate other traces. Some duplication may be necessary, because some reports may have to appear in several traces. An event report may be of interest to two processing agents, for example, and therefore stored in each of their trace-buffers.

Validation of Monitoring Information

Performing validation and plausibility tests on monitoring information to make sure that the system has been monitored correctly is another important monitoring activity. For example, an event may be discarded if the event ID is not an expected one or the value of the time-stamp is not within an acceptable range. Monitoring reports may also be validated in relation to one another: to see whether two event reports in a trace satisfy an expected temporal ordering, for instance; or to check the validity of an event report against the current system status, before applying the status change to the model (more on this in the next section). The detection of invalid orderings may be followed by reordering or filtering of the reports. Validation is performed according to certain validation rules, and a validation report may be generated as in the SIMPLE monitoring system [17].

Database Updating

Valid monitoring information is used to maintain and update a representation or model of the current status of a system. This model is called the Management Information Base (MIB) in OSI terminology. The MIB can be employed by users, managers, or processing agents (for example, by a configuration manager to detect component failures). A data-oriented approach to monitoring communication networks is described in [18]. A conceptual database model of the network is constructed and continuously updated to represent the current status of the network. Some approaches use temporal and historical databases in order to maintain both the current and the historical behavior of the system [10, 19, 20].

There are two general approaches to collecting MIB data. In the dynamic approach, user queries result in the automatic activation of relevant sensors in monitored objects to collect the required data [20]. The advantage of this approach is that only the requested data is collected, reducing processing overheads. The disadvantage is that the queries must be specified before the data is collected. Users, however, may not know in advance exactly what information is required.

On the other hand, in a static approach the collection of data is independent of its use. All possible monitoring data is to be collected and stored for potential access by users. This is done by permanently enabling all sensors or forcing each sensor to be enabled manually. This solves the problem of prespecifying data to be collected but results in the collection of large amounts of information that may not be used.

Because of factors in distributed systems such as communication delays and component failures, it is usually impossible to construct a model that can provide a truly up-to-date and consistent snapshot of the system status. Although a database may be useful for storing trace information and relatively static configuration information, it is not appropriate for holding dynamic status information, which may change in time scales of milliseconds.

Combination of Monitoring Information

Combination (also referred to as correlation or clustering) of monitoring information is the process of increasing the level of abstraction of monitoring data. In conjunction with filtering, it prevents users of such information from being overwhelmed by the considerable volume of details present in a system's activity, so that they can observe the behavior of the system at a desired level of detail. To do this, low-level primitive events and states are processed and interpreted to give a higher-level view of complex states and events. The ability to combine monitoring information is particularly important in distributed systems that implement fault tolerance.

Combination is performed according to event and state definitions, which specify new events and states based on primitive or combined events and states. Events and states at one abstraction level can be used to generate those at higher abstraction levels. When objects are distributed across several nodes, the task of combining monitoring data is more complicated. Local monitoring agents at each node cooperate with one another to detect a global event or state. Each local agent must be told what events and states it should monitor and to which other agents information must be disseminated.

Several languages have been developed that enable users to define new events and states and combine monitoring information. In the State and Event Specification Language (SESL) [21], users define two declarative statements:

e WHEN elist — event e occurs when elist occurs

s EQUALS expr — state s will have the value of expr

An event list (elist) defines a pattern of occurrence of events, which is specified in terms of other event lists, events, low-level events (represented by an event notification message), and state changes. These are related by temporal operators => and ->, and the logical operators | and ! (Table 1). The event list occurs each time a sequence of events that matches the pattern occurs.

Conditions and expressions are defined using the operators listed in Table 2. A condition may also be an expression with a zero value representing true and a non-zero value representing false.

The following examples of SESL script check the availability of a service consisting of two servers:

```
busy       WHEN EVENT ("received
             request")
free       WHEN EVENT ("sending reply")
non_busy   WHEN      one_busy => free
one_busy   WHEN      non_busy => busy |
                     two_busy => free
two_busy   WHEN      one_busy => busy
```

The first two SESL statements are used to show when the server is busy and when it is free. The internal events free and busy are defined in terms

$state	Value of state defined by EQUALS
#event	Number of times event occurred
State("string")	Value of state string of monitoring activity
constant	A numerical constant
!expr -expr	Unary operators
* / + -	Binary arithmetic operators
< > <= >= == !=	Binary relational operators
& |	Binary logical operators

■ **Table 2.** *SESL operators for creating expressions and conditions.*

of external "received request" and "sending reply" events, respectively. The non_busy statement is the internal event associated with the state transition where no server is being used; it is triggered by the event of one server becoming busy followed (without any other events) by the detection of a server becoming free. More details about SESL can be found in [22].

A specification language for defining process level events that can be used for debugging and performance monitoring is described in [23]. A user or programmer includes, with the application, a monitoring section that defines primitive and combined events. This is similar to the declaration of data structures and procedures for an application program. Examples of some event definitions are:

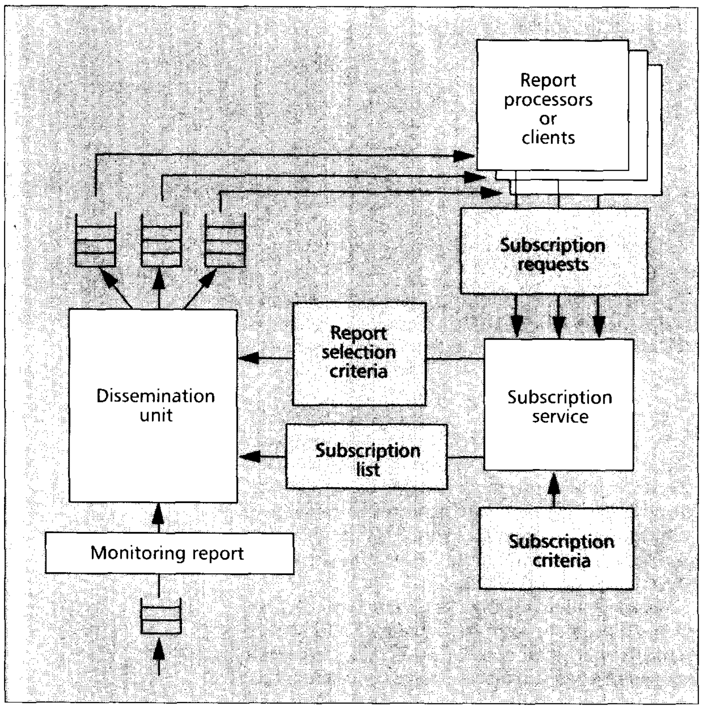
```
e1 ::=  (xmitregister == 1)          on node 0;
e2 ::=  (rcvregister == 2)           on node 0;
e3 ::=  e1 && (waitregister == 1)    on node 0;
e4 ::=  e2 && (waitregister == 1)    on node 0;
e5 ::=  e3 && e4;
e6 ::=  (ptr > 0xE4000)              on node 2;
e7 ::=  (touch(flag))                on node 1;
e8 ::=  (reach(label_1)              on any node;
e9 ::=  (done_flag == TRUE)          on all nodes;
```

Events e1 through e6 are self-explanatory. They are defined in terms of counters (such as rcvregister, xmitregister, and waitregister), a pointer (ptr), and other events on specific nodes. In the definition of e7, the touch() operator specifies events that correspond to any change of a specified variable regardless of the value stored (such as "flag"). In e8, the reach() operator is used for tracing the flow of control of a thread. Users place labels in appropriate sections of the application and define events that correspond to the program counter reaching those points (in this case, label_1) during execution. The definition of an event can span more than one node in the target system (any node, all nodes, or a number of nodes). Users specify various temporal, relational, and logical relationships between events.

Another approach uses a data-manipulation language based on SQL, with some enhancements to specify primitive and combined events [18]. The Event Definition Language (EDL) allows the user to define primitive and higher level events with various filtering constraints [6].

Filtering and Analysis

Typical distributed systems generate large amounts of monitoring information. This results in heavy



■ **Figure 6.** *Dissemination of monitoring reports.*

usage of resources such as CPU and communication bandwidth for generation, collection, processing, and presentation of monitoring information. In addition, users of the monitoring information may be overwhelmed with vast amounts of data which they are unable to comprehend. Filtering is the process of minimizing the amount of monitoring data so that users only receive desired data at a suitable level of detail. Filtering is also needed for security, where certain users should not have access to particular monitoring information.

Filtering functionality must be considered separately from the process of combination of monitoring information: filtering discards information; combining information permits both high-level and low-level views of the information (the information is not discarded). Filtering may be performed explicitly or implicitly in different places and at various stages:

Global filtering: performed by discarding the monitoring reports or traces that do not satisfy global filtering criteria. This includes validation failures.

Reducing report contents: with a variable report structure and the use of a selection facility, a monitoring object receives a report and generates a new one with only a subset of monitoring information contained in the old report. The old report is discarded and the new one used or stored by the object itself or forwarded to another object.

The best policy is to avoid generating unwanted or unnecessary information by:

Controlling report contents: using event and status reporting criteria at generation stage so that only the required information is included in each report.

Conditional generation: A monitoring report or trace is generated when certain predefined conditions are satisfied. In periodic status report generation, for example, increasing the sampling period reduces the frequency and number of generated reports. Report generation mechanisms can be activated or deactivated, and this can be done at various levels of granularity. For example, the reporting mechanisms for all or individual events associated with a component may be deactivated.

Dissemination filtering: disseminating monitoring reports based on a subscriber/provider principle performs an implicit filtering function. Selected reports are forwarded only to those subscribers who have requested them. A monitoring trace may be generated for each destination object. A report may not be discarded, but simply placed in one trace-buffer and not the others. Not including a report in a trace is equivalent to filtering it out from the point of view of the users who have access only to that trace.

It is better to perform the necessary filtering at an early stage to reduce the resource usage in subsequent stages. In all the above cases, implicit or explicit filtering criteria may be based on the information contained within the reports (such as event type, time, priority, and type) or external information such as previous status or events and the capacity to process each report. Some approaches provide a language in which users can define filtering criteria. The OSI Event Forwarding Discriminators perform a combined filtering and dissemination function by selecting event reports based on a logical expression of attributes in the event message. Selected event reports are then forwarded to managers who require them [11].

Monitoring information can be analyzed to determine average or mean variance values of particular status variables. Analysis can range from very simple gathering of statistics to very sophisticated model based analysis. Trend analysis is important for forecasting faults in components. Diagnosis of faults requires correlation of event reports. Some aspects of analysis are considered part of the presentation of information, such as displaying information as histograms or graphs. In general, because analysis is application-specific, it is not considered part of a generalized monitoring service.

Most commercial network monitoring platforms provide facilities for analysis of statistics and display of graphs or histograms. A more complex approach has been adopted in the event-based behavioral abstraction approach (EBBA), which is a paradigm for high-level debugging of distributed systems [6]. The EBBA toolset allows users to construct models of system behavior in a top-down manner. These models reflect user understanding of the expected system behavior and are compared to the actual system activity represented by the monitoring information.

Dissemination and Presentation

Monitoring reports generated by objects are forwarded to different users of such information: human users, managers, other monitoring objects, or processing entities. Dissemination schemes range from very simple and fixed to very complex and specialized. An example of a fixed scheme is the broadcasting of reports to all of a system's users. An example of a complex dissemination scheme is one based on the subscription principle, as shown in Fig. 6 [4].

Clients of a monitoring service subscribe to a dissemination unit by registering themselves with the subscription service. Each client sends a subscription request indicating their identity, the reports required, and the frequency of delivery. Subscription authorization information, held by the service, is used to determine whether clients are authorized users and what reports they are permitted to receive. Selection criteria contained within the subscription request are used by the dissemination system to determine which reports and their contents should be sent to the clients. This provides implicit filtering, because only the requested reports are forwarded.

The OSF distributed management environment (DME) event distribution system is based on the OSI management model. Dynamic subscription is accomplished by creating an event forwarding discriminator (EFD) object, which is actually a thread. This filters and forwards events of interest to a particular manager. An EFD sends reports to a single destination at any time [24]. The EFD is a managed object and so itself can generate notifications which are forwarded by another local EFD.

Generated, collected, and processed monitoring information has to be presented to clients in a format that meets their specific application requirements. A suitable user interface must enable users to specify how to display information, as well as cope with:

- Large amounts of monitoring data generated by the system.
- Various levels of abstraction of such information.
- The inherent parallelism in the system activity, represented by monitoring data.
- The rate at which this information is produced and presented.

Several presentation techniques have been used for displaying debugging data in parallel debugging systems [15]. Similar techniques can be used in generalized monitoring systems to display configuration, performance, security, accounting, and other information. These approaches include the following.

Textual Data Presentation — This is the most common type of display: a simple text presentation of the monitoring information, which may involve highlighting or color (e.g., the Jade monitoring system, for instance [1]). Events may be displayed in their causal rather than temporal order [25]. Appropriate indentation, highlighting, and coloring is used to increase the expressive power of visualization and to distinguish monitoring information at various levels of abstraction. The advantages of this technique are that no special devices are necessary to present monitoring data and that it is simple to convert such information to textual form. However, this technique is not enough to present parallel system activities. Nowadays, simple textual presentation is often used in combination with other more expressive techniques as outlined below.

Time Process Diagrams — The state of a parallel system is represented as a two-dimensional diagram, with one axis representing the objects and the other representing time, that shows the current status of the system and the sequence of events that led to that status. The unit of time may be the occurrence of an event or a period of real time. One or two characters can be used to represent each of the possible events associated with an object or group of objects. The advantage of using a time-process diagram is that monitoring information can be presented on a simple text screen. Time-process display tools may use graphics. In an interactive distributed debugger (IDD), for example, two points in the display are connected by a line to indicate the exchange of a message, instead of placing one character at each point in the display [26]. Users can magnify or scroll to see only a selected portion of the display and select filters to limit the information displayed on the screen.

Time process diagrams usually require a global clock. In one approach using a concurrency map [27], however, events are arranged to show only the order in which they occurred based on causal ordering of a logical clock, instead of showing the exact times of event occurrences based on a global clock.

Animation — The observation of the instantaneous state of the system. A representation of every object or selected portions of monitoring data are placed at a different point in a two-dimensional display. The entire display represents a snapshot of system activity. Such a representation may be in the form of icons, boxes, Kiviat diagrams, bar charts, dials, X-Y plots, matrix views, curves, pie graphs, meters, and so on. Subsequent changes in the display over a period of time provide an animated view of the evolution of the system state. For animation purposes, heavier use of graphics provides the user with more expressive and easily understood views of system activity. Examples of approaches that use animation are SMART and VISIMON tools [28], Radar [7], and the Test and Measurement Processor (TMP) [29].

Because no single view is always sufficient for monitoring purposes, a combination of presentation techniques is often used. Such a general-purpose user interface must allow the criteria that follow.

Visualization at Different Abstraction Levels — A general-purpose user interface must enable users to observe system behavior at a desired level of abstraction. The stepwise refinement, usually used in software engineering, should also be applied to monitoring [30]. Users must be able to start the observation at a coarse level (such as the entire system) and progressively focus on lower levels (subsystems, processes, procedures). This can be achieved by using the combination and filtering techniques discussed previously. Some of the tools that provide this feature are TMP [29] and ConicDraw [31]. Observation of system activities need not be restricted to exactly one level of abstraction at a time, and it is useful to allow users to observe an activity at several levels simultaneously (as in the Demon system [32]). Demon receives messages from various sources such as a live network or previously recorded files, and perform monitoring functionality such as filtering, correlation, analysis, and presentation.

Placement of Monitoring Information — The ability to place a portion of monitoring data on a selected part of the screen can greatly enhance the visibility of the information and aid in com-

■ ■ ■ ■ ■

The advantage of using a time-process diagram is that monitoring information can be presented on a simple text screen.

■ ■ ■ ■ ■

The way that a monitoring system identifies the occurrence of events is an important parameter by which its intrusiveness can be measured.

prehension of potentially very cluttered display. In the time process diagram of IDD, users can move the rows so that information about the related processes can be placed close together. ConicDraw allows users to interact with the tool to improve the visual layout of the display by moving or resizing the boxes representing components and moving the ports so that the lines representing port bindings do not cross.

Controlling the Time of Display — Some display tools provide a history function, permitting users to scroll the display forward or backward in time and control the speed at which the behavior of the system is observed. Display options include start, stop, interrupt, restart, step-by-step display, and continuous display (real-time or slow-motion). Some of the tools which provide this feature are SIMPLE [28] and Radar [7].

Use of Multiple Views — As mentioned previously, multiple views of system activity may be needed to enable the user to obtain a comprehensive picture of system behavior. This can be achieved by using multiple windows presenting the system activities from different points of view (as in the Voyeur system [33]) or different levels of abstraction.

Visibility of Interactions — Some tools such as Radar enable users to display the contents of a particular message [7]). It is useful to be able to chose an event or status report and display its contents, such as its timestamp, object identifier, and so forth. The width or color of the lines representing the components' interconnection could represent the volume of communication between them [29].

Implementation Issues

Intrusiveness of Monitoring Systems

Intrusiveness results from the monitoring system sharing resources (processing power, communication channels, and storage space) with the observed system. Intrusive monitors may alter the timing of events in the system in an arbitrary manner and can lead to:

- Degradation of system performance.
- A change of global ordering of these events.
- Incorrect results.
- An increase in the application's execution.
- Masking or creating deadlock situations.

With an intrusive monitor, observations can only be taken as an approximations of what happens in an unmonitored system [23].

The way that a monitoring system identifies the occurrence of events is an important parameter by which its intrusiveness can be measured. Various detection mechanisms are available and, according to which mechanism is used, monitoring systems can be categorized into three types: hardware monitors, software monitors, and hybrid monitors.

Hardware monitors are separate objects that are used to detect events associated with an object or group of objects. The detection is performed by observation of system buses or by using physical probes connected to the processors, memory ports, or I/O channels [9, 34].

Hardware monitors have the advantage of being nonintrusive, because the resources used by the monitoring system are separated from those used by the monitored system, so that the monitoring system has little or no effect on the observed system. This is particularly important for monitoring real-time systems. Hardware monitors have been successfully used for monitoring communication networks, where a great deal of information is generated and processed rapidly. Hardware monitors, however provide very low-level data which requires considerable processing to provide application level monitoring information. They form the least portable class of monitoring mechanisms, and their installation requires great expertise and thorough knowledge of the system [35].

Nowadays, the design of hardware processor monitors are greatly complicated by the use of pipelining and on-chip caches to increase the throughput of microprocessors and also an increase in the integration of various functional units (such as floating point units and memory management units), which makes monitoring difficult. In the future, this will lead to integration of monitors on the chip [5].

Software monitors usually share the necessary resources with the monitored system. The program is instrumented by inserting software probes in the code to gather information of interest. The source code, library routines, object code, and kernel may be instrumented, manually or automatically, to obtain the required data [14].

Software monitors present information in an application-oriented manner that is easy to understand and use, compared to the low-level information generated by hardware monitors. Software monitors can easily be replicated and are more flexible, portable, and easier to design and construct than hardware monitors.

The disadvantage of software monitors is that they usually use the same resources as the monitored system and therefore interfere in both the timing and space of the observed system, which impacts on its behavior. This impact increases if monitored data is processed and displayed on-line. For this reason, pure software monitors are not adequate for on-line, real-time monitoring. To limit the effect of intrusion, instrumentation must be limited to those events whose observation is considered essential.

Source code can be instrumented in a manual fashion, as in the Meta system [36]. Lumpp*et al.* describe an event-specification language to include a monitoring segment with the source code [23]. The segment is used by the compiler to automatically perform the instrumentation. In the SIMPLE environment, source code instrumentation is performed automatically by using a tool called AICOS [30]. In the Jade system, instrumented library routines are used to monitor the system [1].

Hybrid monitors are designed to employ the advantages of both hardware and software monitors, while overcoming their inefficiencies. Hybrids have their own independent resources but also share some of the resources with the monitored system. Typical hybrid systems consist of an independent hardware device that receives monitoring information generated by software probes inserted into monitored software objects. The event reports generated are processed and displayed by dedicated hardware.

The main advantage of hybrid monitors is that they introduce less intrusion in the monitored system compared with pure software monitors. Like software monitors, they generate high-level application oriented monitoring information, compared with low-level data generated in a purely hardware monitor. It gives them the same flexibility as software monitors. They are also cheaper than hardware monitors, as they share their resources with the monitored system and therefore use less dedicated facilities. Because of this sharing of resources, they are more intrusive than hardware monitors. Hybrid monitors are less portable than software monitors because of their use of dedicated hardware.

Many monitoring systems use the hybrid approach. An example of that is TMP, which allows measuring, monitoring, testing, and debugging of distributed applications [35]. The designers of TMP claim that the degradation in the performance of the monitored system is less than 0.1 percent. The TOPSYS environment supports software, hardware, and hybrid monitoring [51]. ZM4 supports both hybrid and hardware monitoring [17].

Global State, Time, and Ordering of Events

A typical loosely-coupled distributed system consists of a number of independent and cooperating nodes that communicate through message-passing, with no shared memory or common clock. Every node has its own local clock that can be used to timestamp events that occur at that site. Distributed systems are more difficult to design, construct, and monitor, than centralized systems because of parallelism among processors, random and non-negligible communication delays, partial failures, and the lack of global synchronized time.

These features can affect both the behavior of the system and the way it is monitored. Several executions of the same distributed algorithm may result in different interleavings of events and, therefore, various outcomes. This makes the behavior of the system nondeterministic and unpredictable. Furthermore, arbitrary message delays makes it impossible to obtain an instantaneous and consistent "snapshot" view of the system. The same execution of a distributed program may be observed differently by various observers because of different interleavings of monitoring reports. Lack of global time makes it difficult to determine causal relationships between events by analyzing monitoring traces.

One solution is to provide a form of physical clock synchronization based on the exchange of messages containing timestamps, which may contain an external timestamp received from an accurate radio time signal or local time-stamps. The nodes try to maintain processor clocks within some maximum deviation of each other [37, 39]. Alternatively, a system of logical clocks, based on the causality relation, can be used to establish a partial ordering of events in the system [37].

Object Based Implementation

We are using the model described at the beginning of the article as the basis for implementing a generalized monitoring system within the ESPRIT SysMan project. Managed objects will generate monitored information that will be processed by monitoring objects defined using an event manipulation notation similar to that previously discussed in this article to combine, filter, and disseminate events and status information. These monitoring objects can be placed locally to the managed objects and disseminate monitored information to human operators for display via a graphical interface or to automated manager objects. Manager objects require monitored information to make the management decisions which result in control actions to change the behavior of the system being managed. We are considering large scale inter-organizational systems with different managers requiring different information from the managed objects in the system. The monitoring objects can be arranged in a hierarchy to process the information into the required level of abstraction needed by various clients.

The monitoring objects are thus distributed around the system and have management interfaces to permit monitoring to be enabled or disabled and to permit dynamic specification of events [38].

■ ■ ■ ■ ■

The authors are using the model described here as the basis for implementing a generalized monitoring system within the ESPRIT SysMan project.

Conclusion

Generic monitoring services are important tools for managing distributed systems and for debugging during system development. Monitoring services are also essential for debugging during system development and it may be needed as part of the application itself, e.g., process control and factory automation.

We have defined a monitoring model in terms of a set of monitoring functions and as a reference model for explaining the alternative approaches to monitoring distributed systems described in the literature. The main monitoring functions described are: generation of monitored information, which includes status and event reports and traces; processing of monitored information to validate, combine, and filter so that only relevant information is provided to clients; dissemination of required information to those clients who have subscribed to the service and specified the particular information they require; and presentation of monitored information to human users via flexible graphical facilities in a form that aids comprehension and meets specific application requirements.

Acknowledgments

We gratefully acknowledge the support of the SERC and HP Laboratories, Bristol, for a CASE studentship and to the Commission of the European Community (CEC) for funding of the SysMan Project (No. 7026) within the ESPRIT Framework . We also acknowledge the support of our colleagues in the Distributed Software Engineering Section at Imperial College for comments on the concepts described in this paper.

References

[1] J. Joyce *et al.*, "Monitoring Distributed Systems," *ACM Trans. Comput. Syst.*, vol. 5, no. 2, May 1987, pp. 121-50.

[2] M. Sloman, "Distributed Systems Management," Imperial College Research Report, DOC 87/6, April 1, 1987.

[3] L. LaBarre, "Management By Exception: OSI Event Generation, Reporting, and Logging," The MITRE Corporation, 2nd IFIP Symposium on Integrated Network Management, Washington, April 1991, pp. 227-242.

[4] L. Feldkuhn and J. Erickson, "Event Management as a Common Functional Area of Open Systems Management," *Proc. IFIP Symp. on Integrated Network Management*, Boston 1989, pp. 365-76 (North-Holland).

[5] T. Bemmerl, R. Lindhof, and T. Treml, "The Distributed Monitor System of TOPSYS," *Proceedings CONPAR 1990 - VAPP IV*, Sep. 1990, Springer-Verlag, pp. 756-65.

■ ■ ■ ■ ■

Generic monitoring services are important tools for managing distributed systems and for debugging during system development.

[6] P. Bates, "Distributed Debugging Tools for Heterogeneous Distributed Systems," *Proc. 8th International Conference on Distributed Computing Systems*, IEEE, June 1988, pp. 308-16.

[7] R. J. LeBlanc and A. D. Robbins, "Event-Driven Monitoring of Distributed Programs," *Proc. 5th International Conference on Distributed Computing Systems*, May 1985, pp. 515-22.

[8] "Information Technology - Open Systems Interconnection - Systems Management Part 2: State Management Function," ISO/IEC DIS 10164-2 Oct. 1990.

[9] D. Marinescu *et al.*, "Models for Monitoring and Debugging Tools for Parallel and Distributed Software," *Journal of Parallel Distributed Computing* 9, 2, June 1990, pp. 171-84 (special issue: software tools for parallel programming and visualisation).

[10] A. Shvartsman, *"An Historical Object Base in an Enterprise Management Director, Integrated Nework Managment III (C-12),"* Hegering H.-G. and Yemini Y. eds., pp. 123-36 (North-Holland, 1993).

[11] "Information Technology - Open Systems Interconnection - Systems Management Part 5: Event Report Management Function," ISO/IEC DIS 10164-5, Oct. 1990.

[12] P. Dasgupta, "A Probe-based Monitoring Scheme for an Object-oriented, Distributed Operating System," *ACM Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1986, pp. 57-66.

[13] B. Mohr, "Performance Evaluation of Parallel Programs in Parallel and Distributed Systems," *Proceedings CONPAR 1990 - VAPP IV*, Sep. 1990,, pp. 176-87, Springer-Verlag.

[14] M. Van Riek and B. Tourancheau, "A General Approach to the Monitoring of Distributed Memory Machines - A Survey, Laboratoire de l'Informatique du Parallelisme," Ecole Normale Superieure de Lyon, Institut des Sciences de la Matiere de l'Universite Claude Bernard de Lyon, Institut IMAG, Unite de Recherche Associee au CNRS no. 1398, Research Report no. 91/28, September 1991.

[15] C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, vol. 21, no. 4, Dec. 1989.

[16] C. J. Fidge, "Partial Orders for Parallel Debugging," *Proc. Workshop on Parallel and Distributed Debugging*, ACM, 1988, pp. 183-194.

[17] R. Hofmann *et al.*, "Distributed Performance Monitoring: Methods, Tools, and Applications," University of Erlangen-Numberg, IMMD VII, Martensstrabe 3, D-8520 Erlangen, Germany, 1992.

[18] O. Wolfson, S. Sengupta, and Y. Yemini, "Managing Communication Networks by Monitoring Databases," *IEEE Trans. on Sofware Eng.*, vol. 17, no. 9, Sept. 1991.

[19] Y. C. Shim and C. V. Ramamoorthy, "Monitoring and Control of Distributed Systems," *Proc. First International Conference on Systems Integration*, Morristown, NJ, IEEE Computing Press, April 1990, pp. 672-681.

[20] R. Snodgrass, "A Relational Approach to Monitoring Complex Systems," *ACM Trans. on Computer Systems*, vol., 6, no. 2, pp. 157-196, May 1988.

[21] D. Holden, "Predictive Languages for Management," *Proc. IFIP Symp. on Integrated Network Management*, Boston 1989 , pp. 585-96 (North-Holland, 1989).

[22] D. B. Holden, "A Tutorial to Writing Programs in SESL," Internal Report DMP/55, Sys. & S/W Eng. Grp., AEA Industrial Technology, Harwell, Jan. 1991.

[23] J. E. Lumpp, Jr. *et al.*, "Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems," *Proc. 10th International Conference on Distributed Systems*, June 1990, pp. 476-83.

[24] L. Ferrante and M. Santifaller, "DME Event Services Architecture," Document DME 92.08.04, Nov. 1992

[25] C. R. Manning, "Traveler: The Apiary Observatory," *Proceedings of European Conference on Object Oriented Programming*, pp. 97-105, 1987.

[26] P. K. Harter, D. M. Heimbigner, and R. King, "IDD: An Interactive Distributed Debugger," *Proc. 5th Int. Conf. on Distributed Computing Systems*, Denver, Colo., pp. 498-506, (IEEE, May 1985).

[27] J. M. Stone, "A Graphical Representation of Concurrent Processes," *Proc. Workshop on Parallel and Distributed Debugging*, ACM Published as SIGPLAN Notices 24, 1, Jan. 1989, pp. 226-235.

[28] B. Mohr, "SIMPLE: a Performance Evaluation Tool Environment for Parallel and Distributed Systems," A. Bode, editor, *Proc. of the 2nd European Distributed Memory Computing Conference, EDMCC2*, pages 80-89, Munich, Germany, April 1991. (Springer, Berlin, LNCS 487).

[29] D. Haban and D. Wybranietz, "A Hybrid Monitor for Behaviour and Performance Analysis of Distributed Systems," *IEEE Trans. on Software Eng.*, vol. 16, no. 2, Feb. 1990.

[30] R. Klar, A. Quick, and F. Soetz, "Tools for a Model-driven Instrumentation for Monitoring," G. Balbo, editor, *Proc. of the 5th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Torino, Italy, pages 165-180. (Elsevier Science Publisher B.V., 1992).

[31] J. Kramer, J. Magee, and K. Ng, "Graphical Configuration Programming," *IEEE Computing*, Oct. 1989, pp. 53-65.

[32] *"Demon: Distributed Environment Monitoring tool, User's Guide and Reference Manual,"* 1993 (MARI Computer Systems Ltd, Old Town Hall, Tyne and Wear NE8 1HE, UK).

[33] D. Socha, M. L. Bailey, L., and D. Notkin, "Voyeur: Graphical Views of Parallel Programs," *Proc. Workshop on Parallel and Distributed Debugging*, May 5-6, 1988, SIGPLAN NOTICES, 24:1, Jan. 89, pp. 206-215.

[34] J. J.-P. Tsai, K.-Y. Fang, and H.-Y. Chen, "A Non-invasive Architecture to Monitor Real-Time Distributed Systems," *IEEE Computer*, March 1990, pp. 11-23.

[35] D. Wybranietz and D. Haban, *"Monitoring and Measuring Distributed Systems, Performance Instrumentation and Visualization,"* pp. 27-45, (ACM Press, 1990).

[36] K. Marzullo *et al.*, "Tools for Distributed Application Management," Cornell University, *IEEE Computer*, Aug. 1991, pp. 42-51.

[37] L. Lamport, "Time, Clocks and the Ordering of Events in Distributed Systems," *CACM* vol. 21., July. 1978, pp. 558-564.

[38] Y. Hoffner, "The Management of Monitoring in Object-based Distributed Systems," eds. Hegering H, Yemini Y, *IFIP Trans. C12 Integrated Network Management III*, 1993, pp. 235-246.

[39] F. Cristian, "Probabilistic Clock Synchronisation," *Distributed Computing* 3, 1989, pp. 146-58.

Biographies

MORRIS SLOMAN received a Ph.D. in computer science from the University of Essex in the U.K.. He is a reader in the department of computing at Imperial College, London. His research interests include architecture and languages for heterogeneous distributed systems, distributed systems management, and security. He is technical director of the ESPRIT-funded SysMan collaborative project, which is defining domain and policy based tools for managing large inter-organizational distributed systems. He is a coauthor of the textbook *Distributed Systems and Computer Networks* and is coeditor of the *Distributed Systems Engineering Journal*, which is jointly published by IEE, IOP, and BCS. He is a member of the Institution of Electrical Engineers andthe British Computer Society. His e-mail address is m.sloman@doc.ic.ac.uk.

MASOUD MANSOURI-SAMANI received a B.Eng. in computer science in 1991 from Imperial College of Science, Technology, and Medicine, London. He is a research student in the department of computing at Imperial College where he is working on a generalized monitoring service for managing distributed systems. His e-mail address is mm5@doc.ic.ac.uk.