

A revised version of this report has been published as
Chapter 12 of *Network and Distributed Systems Management*,
edited by M. Sloman, Addison Wesley, 1994, pp. 303-347

Monitoring Distributed Systems

(A Survey)

Imperial College Research Report No. DOC92/23

Masoud Mansouri-Samani and Morris Sloman

1 April 1993

Abstract

Monitoring is an essential means for obtaining the information required about the components of a distributed system in order to make management decisions and subsequently control their behaviour. Monitoring is also used to obtain information about component execution and interaction when debugging distributed or parallel systems. This report presents a general functional model of monitoring in terms of generation, processing, dissemination and presentation of information. This model can provide a framework for deriving the facilities required for the design and construction of a generalised monitoring service for distributed systems. A number of approaches to monitoring of distributed systems are compared in the report.

Keywords:

Monitoring, debugging, event reporting, alarm reporting, status reporting, state information, performance management, distributed systems management, network management.

Imperial College of Science Technology and Medicine
Department of Computing
180 Queen's Gate,
London SW7 2BZ,
UK
Email: mm5@doc.ic.ac.uk, mss@doc.ic.ac.uk

CONTENTS

1. Introduction.....	1
1.1 What is Monitoring	1
1.2 Concepts and Terminology	1
1.3 Monitoring Model.....	4
2 Generation of Monitoring Information.....	6
2.1 Status Reporting	6
2.2 Event Detection and Reporting	6
2.3 Trace Generation.....	7
3 Processing of Monitoring Information.....	10
3.1 Merging and Multiple Trace Generation.....	10
3.2 Validation of Monitoring Information	12
3.3 Database Updating.....	12
3.4 Combination of Monitoring Information.....	13
3.5 Filtering of Monitoring Information	15
3.6 Analysis of Monitoring Information.....	16
4 Dissemination of Monitoring Information.....	17
5 Presentation of Monitoring Information	18
5.1 Display Approaches	18
5.2 Desirable User Interface Features.....	22
6 Implementation Issues	25
6.1 Intrusiveness of Monitoring Systems	25
6.1.1 Hardware Monitors	25
6.1.2 Software Monitors	25
6.1.3 Hybrid Monitors	27
6.2 Global State, Time and Ordering of Events.....	27
7 Some Existing Monitoring Systems.....	30
7.1 ZM4/SIMPLE	30
7.1.1 Model-driven Monitoring.....	30
7.1.2 The ZM4/SIMPLE Monitoring Environment	31

7.2 Meta	34
7.2.1 Instrumenting the Application	34
7.2.2 Structure Description	35
7.2.3 Expressing Policy Rules	35
7.3 Demon	35
7.3.1 Event Recognition.....	36
7.3.2 Interpretation.....	36
7.3.3 Graphical Presentation.....	37
7.3.4 Programming Demon.....	38
7.3.5 Start-up Options and Operations	40
7.3.6 Conclusions.....	40
7.4 Monitoring Databases	41
7.4.1 Events.....	41
7.4.2 Trace Collection Service.....	41
7.4.3 Combination	42
8 OSI MANAGEMENT STANDARDS.....	44
8.1 OSI Management Approach	44
8.2 Generation of Monitoring Information	44
8.3 Event Reporting Service	45
8.3 Log Service.....	46
8.4 Processing of Management Information	47
8.5 Discussion	47
9 Summary.....	48
Acknowledgements	48
References.....	49

1. INTRODUCTION

1.1 What is Monitoring

Monitoring can be defined as the process of dynamic collection, interpretation and presentation of information concerning objects or software processes under scrutiny [Joyce et. al 87]. It is needed for various purposes such as debugging, testing, program visualisation and animation. It may also be used for general management activities which have a more permanent and continuous nature (performance management, configuration management, fault management, security management, etc.) [Sloman 87]. In this case the behaviour of the system is observed and monitoring information is gathered. This information is used to make management decisions and perform the appropriate control actions on the system as shown in figure 1.1. Unlike monitoring which is generally a passive process, control actively changes the behaviour of the managed system and in our opinion it has to be considered and modelled separately. In this report we are only concerned with monitoring of object-based distributed systems and particularly its use for management purposes. Note that the generic model of management shown below can be recursively applied to the components of the model itself. As a result a monitoring system itself would have to be managed.

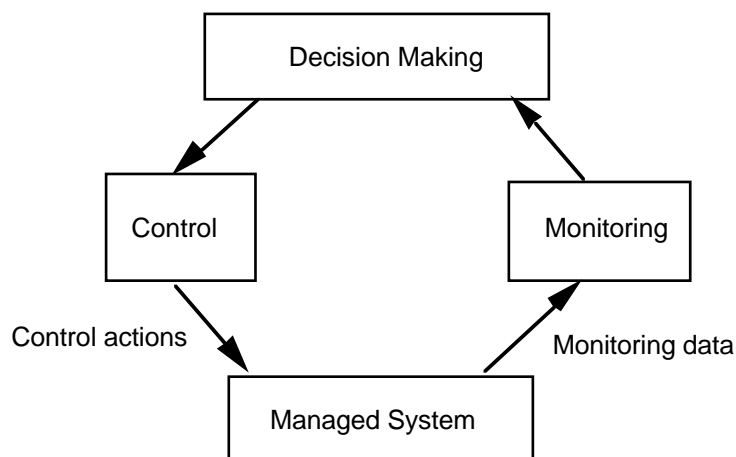


Figure 1.1 Management Model

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 may be out of date. This means it is very difficult to obtain a global, consistent view of all components in a distributed system. Variable delays in reporting events may result in recording events as having occurred in the incorrect order and so some form of clock synchronisation 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 behaviour.

In order to overcome these problems, it is necessary to design a monitoring system in terms of a set of general functions relating to generation, processing, dissemination and presentation of monitoring information. Before we describe this model of monitoring in terms of these activities, we shall introduce some necessary terms and concepts in section 1.2.

1.2 Concepts and Terminology

Since we are considering the subject of monitoring of object-based distributed systems and its use in managing such systems, we must define what we mean by a *managed object*. A managed object is defined as any hardware or software component whose behaviour can be controlled by a management system (from now on we refer to a managed object just as an

object unless explicitly specified). The object encapsulates its behaviour behind an interface which hides the internal details which may be 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 [Sloman 87, Holden 89a], shown in figure 1.2:

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

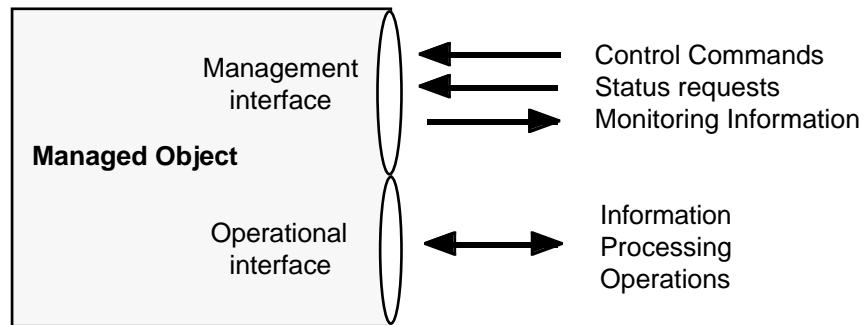


Figure 1.2 A Managed Object

The management interface allows three types of operations:

- i) Control commands (stop, halt, etc.)
- ii) Requests for status information
- iii) Monitoring information generated by the object.

An object may be *passive* or *active*. A passive object (or server) encapsulates some permanent resource, such as a data structure, and a set of routines and operations that can be performed on the resource (cf. monitors). It provides services which are used by one or more active objects or clients. An active object performs some function and may also encapsulate some shared resource and the operations for accessing it, but it may invoke operations on other objects. It should be possible to monitor both active and passive objects.

Monitoring can be performed on an object or a group of related objects (a *monitoring domain*). Each object has associated with it a *status*, and a set of *events* (i.e., status changes). The behaviour 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 behaviour at a discrete point in time and is represented by a set of *status variables* contained within a *status vector* [Feldkuhn & Erickson 89]. These variables or attributes may be *static* (e.g., machine type) or *time-varying* (e.g., load). A static attribute may further be subdivided into those associated with a *permanent* object or a *temporary* object. An *event* is defined as an atomic entity that reflects a change in the status of an object. The status of an object has a duration in time, e.g., "process is idle" or "process running", whereas an event occurs instantaneously, e.g., "message sent" or "process started". Usually, the status of an object is changing continually and therefore the behaviour of the object is normally observed in terms of a distinguished subset of events, called *events of interest*. These reflect changes that are of significance to the management and therefore are generated when a pre-defined set of conditions are satisfied. In a distributed system three kinds of events can be identified [Bemmerl et. al 90]:

i) 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 P2 enters/leaves procedure Fred for the nth time.
- the operating system enters the scheduler.

- ii) 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.

Although such an event is caused by a specific control flow event, it is not associated with any particular control thread.

- iii) *Process-level events* show the creation and deletion of processes and the interactions and data flow between them. For example,
- process P1 started,
 - 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 can be referred to as *internal events* which 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 state which 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 analysing the behaviour of a single component.

Process-level events can be considered as *external events*, which represent the external behaviour of an object and its interactions with other objects. These events are of particular interest to management. A generalised monitoring system should allow us to observe a combination of these events. Many monitoring tools enable the users to specify and detect only process-level events such as interprocess communication, as these events do not violate the encapsulation of objects and are at the correct level of abstraction for analysing the behaviour of distributed systems [Bates 88, Joyce et al. 87, LeBlanc & Robbins 85].

Events can also be classified, according to their level of abstraction, into *primitive* and *combined* (or *correlated*) events.

- i) A primitive event signifies a simple change in the state of an object.
- ii) A combined event is defined as a combination or grouping of other primitive and combined events.

This classification is useful for describing the global behaviour of a group of objects in terms of the local behaviour of every object in the group¹. Various languages are used for specifying combined events and states. This is described in more detail in section 3.4, on combination of monitoring information.

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 behaviour 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 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 generalised

¹ [Holden 88] uses the terms *object set states* and *object set events* to describe the overall behaviour.

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

1.3 Monitoring Model

This survey is based on a general functional model which is derived from the Event Management Model of [Feldkuhn & Erickson 89], with some changes and enhancements. This model identifies the following four monitoring activities performed in a loosely-coupled, object-based distributed system:

- i) *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.
- ii) *Processing*: A generalised monitoring service provides common processing functionalities 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.
- iii) *Dissemination*: Monitoring reports are distributed to users, managers or processing agents who require them.
- iv) *Presentation*: Gathered and processed information is displayed to the users in an appropriate form.

Implementation issues relating to the intrusiveness of the monitoring system which depends on whether it is implemented in hardware or software and how clock synchronisation is achieved for event ordering, provide a fifth dimension for comparing monitoring systems. Figure 1.3 summarises the elements of the monitoring reference model presented in the report.

Many models have been developed in order to describe the monitoring process. One approach has been to identify a set of layers such as the Event/Action Paradigm of [Marinescu et al. 90]. At first sight, the above four activities appear to be a layered model with generation as the lowest layer and presentation using the services of the lower layers. However a generalised monitoring system may need to perform these activities in various places and in different orders to meet specific monitoring requirements. For example generated information may be directly displayed by an object without processing or dissemination. Events and reports which are distributed to particular managers, could 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 which can be combined as required in a generic monitoring service.

We shall describe these activities of the Monitoring Model in detail in sections 2 to 5. Section 6 discusses some issues related to implementation of a monitoring service, such as intrusiveness of monitoring and ordering of events. A brief summary of some of the existing approaches is presented in section 7 and the relevant Open Systems Interconnection Management standards are described in section 8.

Generation of Monitoring Information

- Status reporting
- Event detection and reporting
- Trace generation

Processing of Monitoring Information

- Merging and multiple trace generation
- Validation
- Database updating
- Combination
- Filtering
- Analysis

Dissemination of Monitoring Information

- Registration of subscribers to 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 synchronisation for event ordering

Figure 1.3 Elements of a Monitoring Reference Model

2 GENERATION OF MONITORING INFORMATION

Monitoring data is generated in the form of *status* and *event reports* (figure 2.1). A sequence of such reports is used to generate a *monitoring trace*. Status reporting, event detection and reporting, and trace generation are described below.

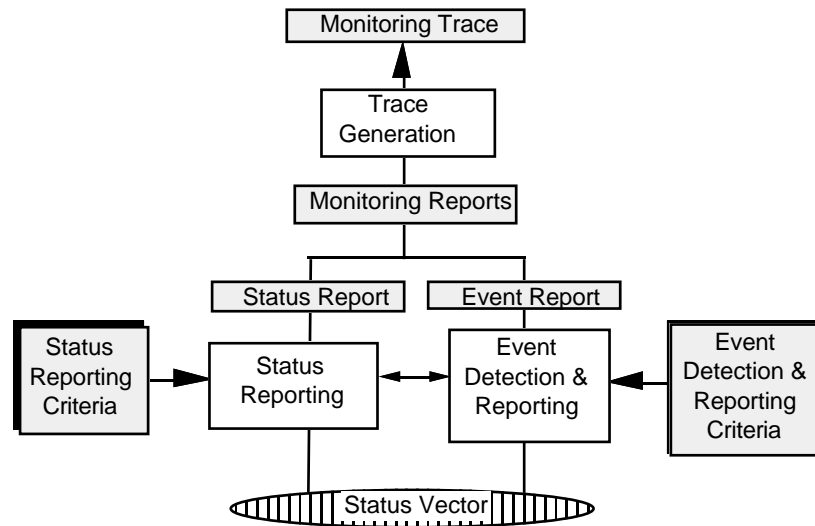


Figure 2.1 Generation of Monitoring Reports and Traces

2.1 Status Reporting

A status report contains a subset of values from the status vector and may include other related information e.g., time stamp and object identity. It represents the status at a specific instance in time and can be generated as described below.

- i) *Periodic*: Reports are generated based on a pre-determined schedule.
- ii) *On request*: A report is generated upon receiving a request (*solicited* reporting). Note that the request may itself be periodic (i.e., polling) or on a random basis.

One example of periodic status reporting can be found in Conic environment [Magee, et al. 89] where each node sends a configuration status report to a name server every 5 seconds. The OSI event management permits scheduling of event reports on both daily and weekly basis [ISO 10164-5].

The Clouds operating system [Dasgupta 86] uses on demand status reporting scheme. Requests are called *probes* and every object or process has a predefined or user defined probe procedure which is executed whenever a probe is received. The probe handler sends a message back to the originator of the request, reporting the status condition of the process.

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

2.2 Event Detection and Reporting

Significant changes in the status of an object or a group of objects (i.e., events of interest) would have to be detected. An event is said to have occurred when certain pre-defined conditions are satisfied (defined by *event detection criteria*). To detect events associated with an object certain software or hardware *probes* or *sensors* would have to be inserted and installed in the object. Insertion of such mechanisms is part of a process called *instrumentation*. Where, how and when event detection is carried out depends on the resources available for detection (e.g., dedicated hardware, communication channels) and the *intrusiveness* of the monitoring

system. This intrusiveness or *probe effect* is the degree to which the observed system is perturbed by the act of monitoring² and is discussed further in section 6.1 on intrusiveness of monitoring systems.

i) Location of Event Detection

Event detection may be internal within the object and typically performed as a function of the object itself. For example, a function that updates the status vector may check the event detection criteria when it performs the update. Event detection may also be performed externally from the object itself e.g., by an external agent which receives status reports and detects changes in the state of the object.

ii) Time of Event Detection

Detection of event occurrences may be *immediate*, (real-time) or *delayed*, detected some time after the occurrence. 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 is particularly used for detection of events in time frames of milliseconds. Alternatively, status reports may be generated, stored and used to detect events at some later time.

iii) Event Report Format

Once the occurrence of an event is detected an event report is generated. It contains *attributes* such as the event identifier, type, priority, time of occurrence, the state of the object immediately before and after the occurrence of the event and other application specific status variables.

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

Obviously, the amount of information contained in a monitoring report depends on the requirements of users or clients who need them. The attributes assigned to each event or status may be of two kind: *Independent* or *Dependent* [Mohr 90]. Independent attributes are those primitive attributes which are assigned to all events and status, such as the time stamp, identity of the object, etc. Dependent attributes are assigned depending on the type of the event or status. For example, a configuration event report, representing a "create process" event may contain the identity of the created process. Event and status reporting criteria is used to determine what information to include within each report.

The format and structure of a report may be *fixed* or *variable*. Obviously with a variable report structure some amount of filtering can be performed implicitly, where only the necessary information is included within a report as explained in section 3.5 on filtering.

2.3 Trace Generation

In order to describe the dynamic behaviour 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.

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 certain period of time. It describes "a completely observed time interval" of the

² Here we only consider *detection intrusion*, associated with the recognition of events, and not *action intrusion*, associated with performing control actions on the system.

behaviour of the monitored object or system. A trace may be segmented due to overflow of a trace buffer, or deliberate halting of trace generation which results in the loss or absence of reports over a period of time.

A trace may have a header giving some general information such as the start and end time stamps, the identity of the monitored object, its size and the identity of the program etc. Monitoring traces may be generated for various reasons:

- *Archiving purposes and post-mortem analysis:* Monitoring reports may be needed at some later stage for further processing, analysis and usage. They are stored as monitoring traces and examined by the user at a later time, possibly after the completion of the program. This is particularly important for debugging purposes [McDowell & Helmbold 89]. Archive traces are usually referred to as *histories* or *logs*. A logging service can be used to generate these traces in log files. Certain facilities or services can be used to browse or query these traces. Such traces can also be used to control a replay or re-execution of the program, allowing the reproduction of the erroneous computations. With a complete trace an individual process can be debugged in isolation, where the history provides the needed communication and simulates the environment of the process.
- *Availability of resources:* Other reasons for forming these traces may be lack of processing power to analyse and interpret the monitoring reports "on the fly"; or limited communication resources to send reports to an external processing agent, as and when they are generated. This is particularly true for real-time monitoring.
- *Speed of visualisation:* Trace generation might be necessary when the rate at which monitoring information is received and displayed is too quick for the observer to follow. This is especially true for real-time observation and display of system activity. Special display tools can be used to control the speed at which information is presented to the user.
- *Transformation of the logical view of the system activity:* It enables the construction of a global monitoring trace from local traces, which describe the behaviour of the system as a whole (see merging of monitoring traces - section 3.1.1). Multiple traces can be generated from a single trace, to reflect specialised or restricted views of the system activity (see generating multiple traces - section 3.1.2). Lower level histories can be used to generate more meaningful higher level monitoring information [McDowell & Helmbold 89], by using combination and filtering (see sections 3.4 and 3.5 respectively).

The design alternatives relating to trace generation include:

- *Location of trace generation:* Traces could be formed by objects which generated the monitoring reports, by intermediate monitoring objects when processing monitored information, or by the final user of such information.
- *Temporary vs. long-term traces:* Temporary storage involves placing the monitoring reports in a temporary buffer before they are processed or transferred to another object for use. Generally a report is removed from short term storage when accessed. Long-term storage involves recording the reports in persistent log or history files by making use of a logging service. The report would not usually be removed if it is read.
- *Storage capacity:* Obviously there is a limit to the amount of available storage space and therefore the size of a monitoring trace. Overwriting older records when the maximum storage size is reached can be appropriate if we are interested in the most recent behaviour of the system. Alternatively new reports are discarded until more space is available [LaBarre 91] by halting trace generation. Both these strategies may result in a segmented trace as some of the reports may be discarded. A capacity threshold event should be generated to indicate storage overflow.
- *Sophistication of trace generation:* A simple scheme would store all the reports, in the arrival order, in a local trace-buffer of a monitoring object, without changing the contents of the stored information. This is usually used for temporary storage of monitoring data.

A sophisticated scheme might use a logging service (which can be modelled as a managed object [LaBarre 91]) for the long-term storage of reports together with some extra

information, in a variety of formats, representations and orderings. It may allow the generation of multiple traces or logs, representing different logical views of system activity (see section 3.1.2). Special log records could be generated containing a log record identifier, logging time, information contained in the report to be logged and other related data. 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 time stamp is available. This overcomes the problems of ordering according to arrival order which may lead to incorrect interleavings of monitoring reports and invalid observations, due to communication delays. Determining the order of monitoring reports is discussed in section 6.2.

- *Access to the trace reports* may be *on demand* by issuing 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 (e.g., generated by a particular object), the trace file from which they may be read or the number of reports needed.

Pre-determined conditions may be used to transfer reports from a trace-buffer to a remote processing agent when the buffer is full, contains n elements, the communication load is below a certain threshold or the processing load is low [Van Riek & Tourancheau 91]. These strategies are particularly useful for reducing the communication overheads by sending reports in blocks and therefore economising on channel set-up time.

3 PROCESSING OF MONITORING INFORMATION

In previous sections we discussed the steps necessary for generation of monitoring information. In this section we will consider some common processing activities that can be performed on this information. A monitoring service could provide certain functional units (as building blocks) which can be combined in different ways to suit the monitoring requirements. Figure 3.1, shows one possible combination. Note that these processing functionalities are often integrated and are performed in different places and at various stages.

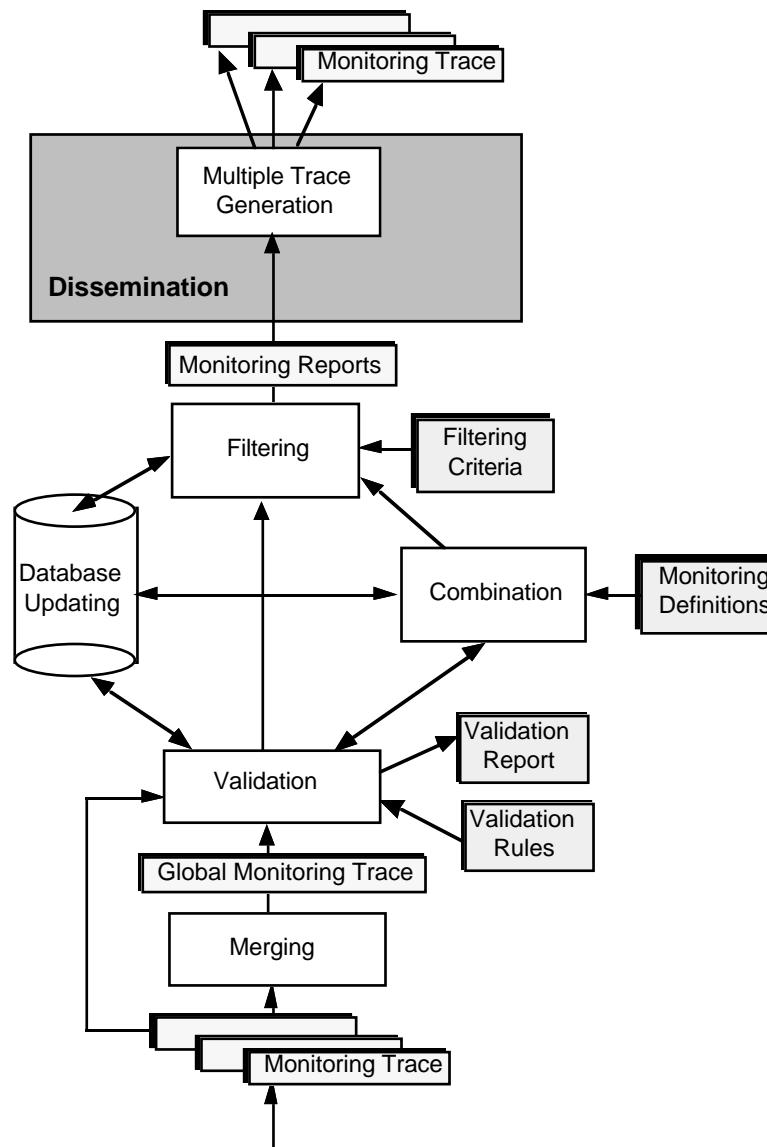


Figure 3.1 Processing of Monitoring Reports

3.1 Merging and Multiple Trace Generation

In order to provide different logical views of system activity over a period of time, monitoring traces may be constructed and ordered in various ways. The attributes of a report which can be used as selection criteria in determining how monitoring traces are processed include:

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

Construction of monitoring traces is performed according to a *trace specification*, based on these factors. It specifies how the final traces are formed and what they will contain. Obviously not including a report 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 described in the following sections.

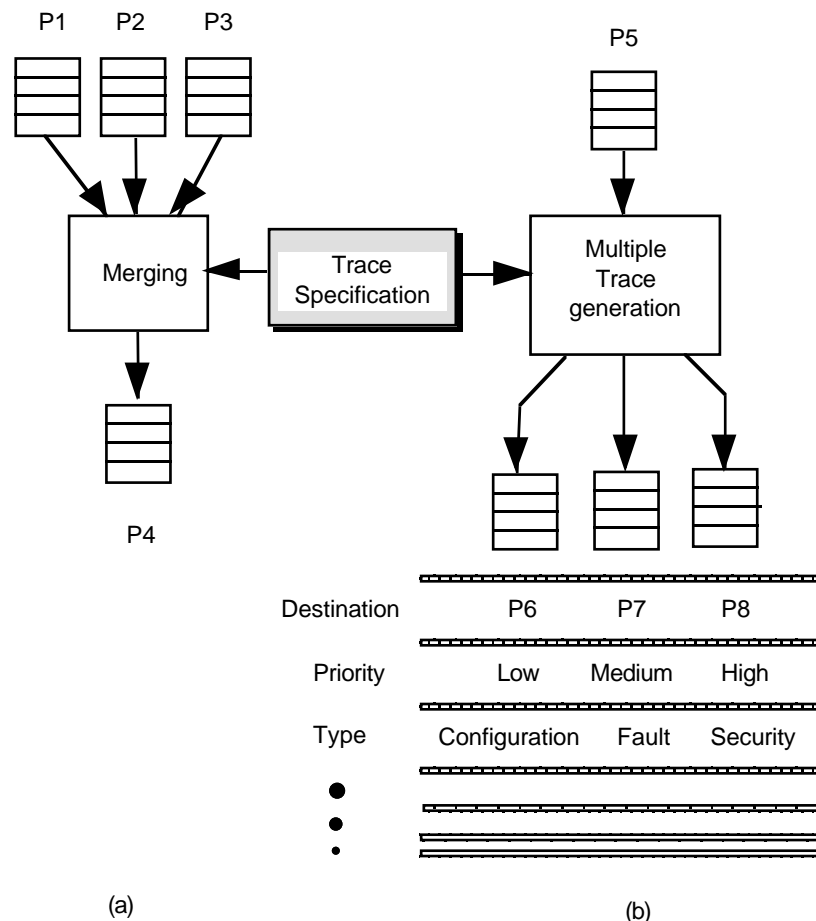


Figure 3.2 Merging and Multiple Trace Generation

3.1.1 Merging of Monitoring Traces

A trace segment containing 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, as shown in figure 3.2a, is merging of several monitoring traces into one trace. For example, if a monitored system consists of three objects P1, P2 and P3, local trace segments from these objects can be merged using scanning and selection services to generate a global trace segment, representing the global behaviour of the system in a particular interval. The trace segment generated can then be used by a processing agent P4.

Merging could be an iterative operation. A trace generated in this way could itself be merged with others to generate a more general trace. The original traces may be discarded once merging is complete. Generating one global monitoring 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 since 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 behaviour of a distributed system [McDowell & Helmbold 89]. A general technique for obtaining the required partial ordering is described in [Fidge 88]. To achieve this a vector of logical time stamps is associated with each event. By comparing the vectors the ordering of events can be determined. This is described in more detail in section 6.2.

3.1.2 Generating Multiple Traces

As shown in figure 3.2b, 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 as shown in figure 3.2. For example, a trace could be generated with all high priority security events, so that they can be processed first and another trace with all the accounting events destined for a particular process. A trace generated in this way may be used to generate other traces.

Obviously some duplication may be necessary because some reports may have to appear in several traces. For example, an event report may be of interest to two processing agents and therefore it is stored in each of their trace-buffers.

3.2 Validation of Monitoring Information

Another important monitoring activity is performing validation and plausibility tests on monitoring information, to make sure that the system has been monitored correctly. This may be performed at different levels. When an individual report is received, its contents may be checked to see if they are valid. For example, whether the event id. represents one of the expected events, or the value of the time-stamp is valid. Invalid reports are discarded. Monitoring reports may also be validated in relation to one another. For example, to see whether two event reports in a trace satisfy an expected temporal ordering, or to check the validity of an event report against the current system status, before applying the status change to the model (see database updating). The detection of invalid orderings may be followed by reordering or filtering of the reports. Validation is done according to certain validation rules, and a validation report may be generated. The SIMPLE monitoring system [Hofmann, et al.92] performs some validation and plausibility tests on selected event reports and event traces (section 7.1).

3.3 Database Updating

Valid monitoring information, may be used to maintain and update a representation or model of the current status of the system. In OSI terminology it is called the *Management Information Base* (MIB). This representation could be used by other users, managers or processing agents. For example, the configuration manager may periodically examine this data model to detect component failures. Section 7.4 describes a data-oriented approach to monitoring communication networks in which a conceptual data-base model of the network is constructed and continuously updated to represent the current status of the network [Wolfson et al. 91]. Some approaches use temporal and historical databases in order to maintain both the current and also the historical behaviour of the system [Shim & Ramamoorthy 90, Snodgrass 88].

There are two general approaches to collecting MIB data. A *dynamic* approach in which only the information which is requested by the user is collected (e.g., [Snodgrass 88]). User queries result in the automatic activation of relevant sensors in monitored objects and the collection of the required data. The advantage of this approach is that only the requested data is collected. This is particularly important when monitoring resources (processors, memory, etc.) are limited and / or there are many sensors present. The disadvantage is that the queries must be specified *before* the data is collected. The user, however, may not know in advance exactly what information is required. At the other extreme, a *static* approach is used in which the collection of data is independent of its use. All possible monitoring data must be collected and stored for potential access by users. This is done by permanently enabling all the sensors, or forcing each sensor to be enabled manually. This solves the problem associated with the

previous approach but results in collection of large amounts of information that may not be used.

In a distributed system, due to various factors such as communication delays and component failures, it is usually impossible to construct a model which could provide a truly up-to-date and consistent snapshot of the system status. This is further discussed in section 6.2.

3.4 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 the users of such information from being overwhelmed by the considerable volume of details present in all of the system's activity, so that they can observe the behaviour of the system at a desired level of detail. To do this low-level primitive events and states can be processed and interpreted to give a higher-level view of the complex states and events which occur in the system. Combination is performed according to certain *event and state definitions*. These definitions specify new events and states based on primitive or other combined events and states. In other words, events and states at one abstraction level can be used to generate those at higher abstraction levels. The ability to combine monitoring information is particularly important in distributed systems that implement fault tolerance. Events generated by several sensors or probes with different failure modes can be combined to provide the necessary reliability and resilience against failures. For example, the expression:

$$t\text{-exceeded} = t_1\text{-exceeded OR } t_2\text{-exceeded OR } t_3\text{-exceeded}$$

with $t_i\text{-exceeded}$ representing the "temperature exceeded" event detected by sensor i , may be used to make sure that an event representing an increase in temperature above a pre-defined limit is detected even if two out of three independent sensors fail.

Various languages have been developed which enable the user to define new events and states and combine monitoring information.

In the State and Event Specification Language (SESL) [Holden 89b], a user can 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 \Rightarrow and \rightarrow , and the logical operators $|$ and $!$ (see table 3.1). Each time a sequence of events occurs which matches the pattern, the event list also occurs.

elist \Rightarrow elist2	matches when elist2 occurs immediately after elist1
elist1 $>$ elist2	matches every time elist2 occurs after elist1 has once occurred
elist1 \rightarrow elist2	matches when elist2 occurs sometime 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 3.1 Example SESL operators for creating event lists

Also various conditions and expressions can be defined using the operators listed in table 3.2. A condition is also an expression with a zero value representing true and a non-zero value representing false.

\$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 3.2 SESL operators for creating expressions and conditions

As an example, to check the availability of a service consisting of two servers, the user can write the following SESL script:

```

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 can be used to show when the server is busy and when it is free. Internal events *free* and *busy* are defined in terms of external events "received request" and "sending reply", respectively. The next three statements define *non_busy*, *one_busy*, and *two_busy* events. For example, *non_busy* is the internal event associated with the state transition where no server is being used, and is triggered by the event of 1 server becoming busy followed (without any other events) by the detection of a server becoming free. More details about SESL can be found in [Holden 91].

A specification language for defining process level events which can be used for debugging and performance monitoring is described in [Lumpp, et al. 90]. The 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 the application program. Examples of some event definitions are shown below:

```

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-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*, *touch()* operator is used which specifies events that correspond to any change of a specified variable regardless of the value stored (e.g., *flag*). In the definition of *e8*, the operator *reach()* is used for tracing the flow of control of a thread. The user can 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 (e.g., any node, all nodes, 5 nodes). The user can specify various temporal, relational and logical relationships between events.

In another approach, [Wolfson et al. 91] use a data manipulation language based on SQL, with some enhancements to specify primitive and combined events (section 7.4). Also, the Event Definition Language (EDL) [Bates 88] allows the user to define primitive and higher level events with various filtering constraints.

3.5 Filtering of Monitoring Information

A typical distributed system may generate large amounts of monitoring information. This results in heavy usage of resources such as CPU and communication bandwidth for generation, collection, processing, and presentation of monitoring information. In addition, the users of the monitoring information may be overwhelmed with vast amounts of data which they are unable to comprehend. Filtering is the process of minimising the amount of monitoring data, so that users only receive desired data at a suitable level of detail relevant to their purposes. It 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, but combining information permits both high level and low level views on the information i.e. 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 which 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 could receive a report and generate a new one with only a subset of monitoring information contained in the old report. The old report could be discarded and the new one may be used or stored by the object itself or forwarded to another object.

Obviously the best policy is to avoid generating unwanted or unnecessary information. For example this could be done by:

- *Controlling report contents*: by 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. For example, in periodic status report generation, by increasing the sampling period, the frequency and number of generated reports can be reduced. Also report generation mechanisms can be *activated* or *deactivated*. This could 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 as selected reports are forwarded only to those subscribers who have requested them as explained in section 4. 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. Obviously, not including a report in a trace would be equivalent to filtering it out from the point of view of the users who have access only to that trace.

Clearly, 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 (e.g., event type, time, priority, type) or external information such as previous status or events and the capacity to process each report. Some approaches provide a language in which various filtering criteria can be defined (e.g., FDL in the SIMPLE environment - section 7.1). Users of monitoring reports should be able to define their own filtering criteria.

3.6 Analysis of Monitoring Information

Monitoring information can be analysed to determine average or mean variance values of particular status variables (see section 8.4). 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 e.g. displaying information as histograms or graphs. In general, analysis is application specific so is not really considered part of a generalised monitoring service. It can range from very simple gathering of statistics to very sophisticated model based analysis.

Some monitoring tools collect various statistics such as total CPU usage, ready, blocked, idle and busy times, number of messages or bytes sent, etc. (TMP [Wybranietz & Haban 90]). In the SIMPLE environment [Mohr 90], a commercial data analysis and graphics package S from AT&T, has been integrated for interactive and complex analysis of monitoring data. A more complex approach has been adopted in the Event Based Behavioural Abstraction approach (EBBA), which is a paradigm for high-level debugging of distributed systems [Bates 88]. The EBBA toolset allows the user to construct models of system behaviour in a top-down manner. These models reflect user understanding of the expected system behaviour, and are compared to the actual system activity represented by the monitoring information.

4 DISSEMINATION OF MONITORING INFORMATION

Monitoring reports generated by the objects would have to be forwarded to different users of such information. The destination of such reports may be human users, managers, other monitoring objects or processing entities. Dissemination schemes range from very simple and fixed to very complex and specialised. An example of a fixed scheme is to broadcast all the reports to all the users. A complex and specialised dissemination scheme could be based on the subscription principle [Feldkuhn & Erickson 89] as shown in figure 4.1.

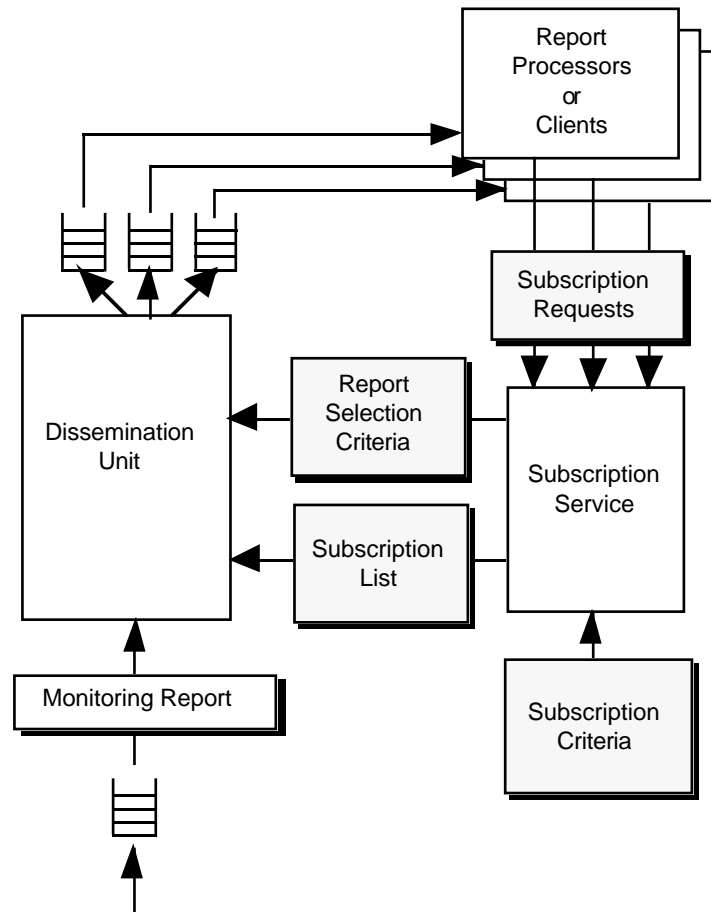


Figure 4.1 Dissemination of monitoring reports

Clients of a monitoring service subscribe to receive the required status or event reports from the dissemination unit by registering themselves with the subscription service. Each client sends a subscription request indicating its identity, the list of reports and the frequency of the reports required. Subscription authorisation information, held by the service, is used to determine whether the client is an authorised user and what reports it is permitted to receive. Only authorised users are permitted to subscribe and are entered in the *Subscription List* maintained by the dissemination unit. *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 as only the requested reports are forwarded.

5 PRESENTATION OF MONITORING INFORMATION

Generated, collected, and processed monitoring information has to be presented to clients in a format which meets their specific application requirements. A suitable user interface should enable the user 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.

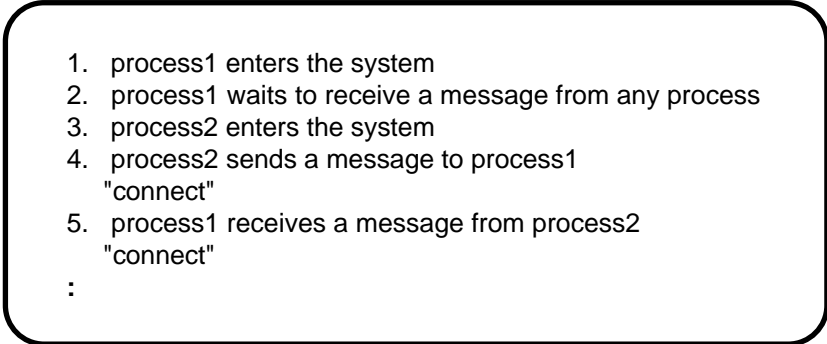
5.1 Display Approaches

In this section, we shall describe different techniques that can be used to display information to the user, and we will conclude by considering useful features of some existing tools. Various presentation techniques have been used for displaying debugging data in parallel debugging systems [McDowell 89]. Similar techniques can be used in a generalised monitoring system to display different types of monitoring information relating to configuration, performance, security, accounting, etc. These approaches are described below:

5.1.1 Textual Data Presentation

This is the most common type of display with a simple text presentation of the monitoring information, which may involve highlighting or colour. Events may be displayed in their *causal* rather than temporal order as in Traveller [Manning 1987].

The Jade monitoring system [Joyce et al. 87] observes the process level events such as interprocess communication, creation and killing of a process. It provides a Text Console which displays one or two lines of textual output to describe the event. It shows the name of the process initiating the event, the event type, the name of the process that is the subject of the event, if any, on the first line. If the event is one in which processes communicate, the contents of the message are printed as the second line of the output. An example of such a display is shown figure 5.1.



```
1. process1 enters the system
2. process1 waits to receive a message from any process
3. process2 enters the system
4. process2 sends a message to process1
   "connect"
5. process1 receives a message from process2
   "connect"
:
```

Figure 5.1 Example Jade Textual Display

Appropriate indentation, highlighting and colouring can be used to increase the expressive power of visualisation and also to distinguish monitoring information at various levels of abstraction. 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.

5.1.2 Time Process Diagrams:

The state of the parallel system is represented as a two dimensional diagram, with one axis representing the objects and the other representing time. It shows the current status of the system and the sequence of events that led to that status, and therefore can display patterns of behaviour over time. 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.

In Jade [Joyce, et al. 87], an *event line console* has been provided which displays the current state and history of each process in a compact form and, at the same time, defines the relative ordering of events, as shown in figure 5.2. The names of the processes and single-letter abbreviations are listed on the right-hand side. In the middle there is one row for each process representing the event line for that process. Each event line is divided into a number of *event intervals* separating adjacent events, and each interval displays an event; events are inserted at the right of the display and scroll to the left. The last event to scroll off the left-hand side is always displayed (to the left of the vertical bar) so that, if a process has had all of its events scrolled off of the middle section, the current status of a process is always available. Here, A dotted line "..." signifies that the process is blocked and a dashed line "---" means that it is executing.

Event	Line	Console	Commands: Go	Pause	Step	Quit
a	<RA			a	P1
b	<RAd>-Rd-----<RA.....			b	P2
c	<RA	f>-----			c	P3
d	<RAe>-<Sb.....>-Re-----<RA.....e>-<Sb			d	P4
e	Sd>	----<Sd>-----<Sd.....			e	P5
f	<Sc			f	P6
g		E-<Sa.....			g	P7
EVENTS:	I	initialise	<Sp...>	send to p	Cp	create p
	E	enter_system	<Rp...>	receive from p	Kp	kill p
	L	leave_system	<RA..p>	receive any	K	killed
	X	exit	Rp	reply to p		

Figure 5.2 Jade's time-process diagram (Event Line Console)

Time-process display tools may use graphics. For example in IDD [Harter et al. 85] 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. The user can magnify or scroll to see only a selected portion of the display. Various filters may be selected by the user to limit the information displayed on the screen.

Time process diagrams usually require a global clock. However, in one approach using a concurrency map [Stone 1988], 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. As shown in figure 5.3, the map displays the process histories as event streams on a time grid. Each column of the grid displays the sequential event stream of a single process. Every row represents an interval of time, and the events that appear in different columns in that row occur concurrently. All the events in one row occur before any of the events in the next row. Time dependencies are expressed by arrows.

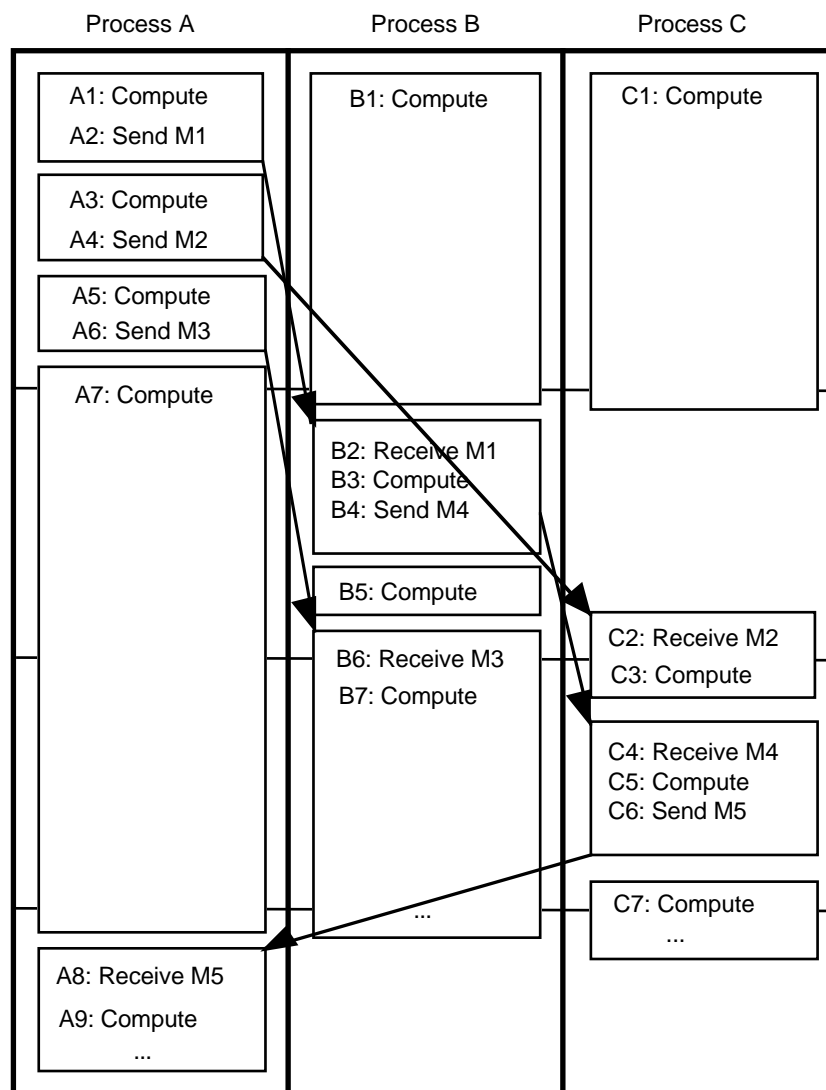


Figure 5.3 Concurrency Map

5.1.3 Animation

Animation allows the observation of the instantaneous state of the system. A representation of every object or selected portions of monitoring data, can be 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, etc. Subsequent changes in the display, over a period of time, could provide an animated view of the evolution of the system state.

In SIMPLE, a visualisation program (called SMART) is provided which can be used on any ASCII-terminal [Mohr 91]. Figure 5.4 shows a screen snapshot of SMART. In this example every column represents a process and the events which can occur in it. The user can step through the event trace by hitting a key, the next event in the trace is highlighted on the screen, and the current time is displayed at the bottom of display. There is also a slow-motion mode for displaying events in a speed proportional to real time.

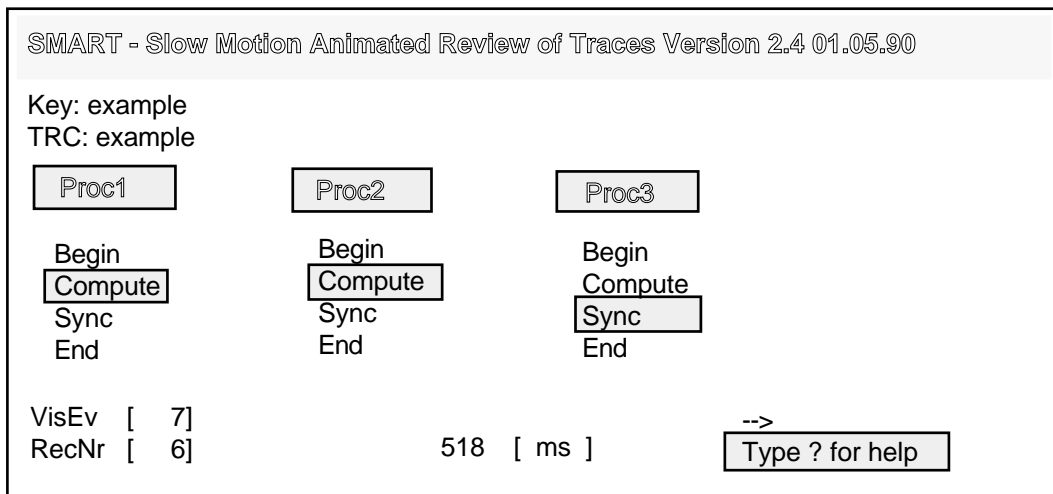


Figure 5.4 Screen Snapshot of SMART

Another visualisation tool has also been provided called VISIMON based on X-Windows with graphics capabilities which allows the user to animate the execution of the program according to a user specified animation description.

The Radar system monitors process level events and uses animation of messages, as shown in figure 5.5 [LeBlanc & Robbins 1985]. It has two windows. The top window shows a textual display of the events which are occurring, and the lower window has a graphical representation of the same events. The user can see processes (represented as boxes) and messages (shown as [+]) queued on their input ports. The drawing of a process indicates the number of input and output ports associated with that process. The user can have the contents of a message displayed and can set the speed of animation to single stepping or continuous.

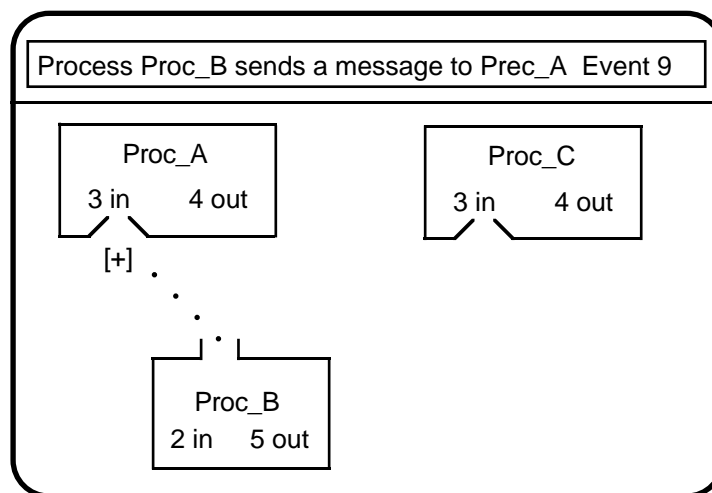


Figure 5.5 The Radar Display

For animation purposes, heavier use of graphics provides the user with more expressive and easily understood views of system activity. For other examples of animation technique see [Haban & Wybranietz 90].

Often a combination of various presentation techniques, may have to be used to provide different views of system activity, because no single view may be sufficient for monitoring purposes. Based on the studied monitoring display tools, we can mention some desirable features which a general purpose user interface must possess.

5.2 Desirable User Interface Features

i) Visualisation at Different Abstraction Levels

A general purpose user interface must enable the user to observe system behaviour at a desired level of abstraction. The stepwise refinement, usually used in software engineering, should also be applied for monitoring [Klar et al. 92]. The user should be able to start the observation at a coarse level and progressively focus on lower levels. Figure 5.6 shows the hierarchical structure of a distributed system including the view of the physical distribution of the program [Wybraniec & Haban 90].

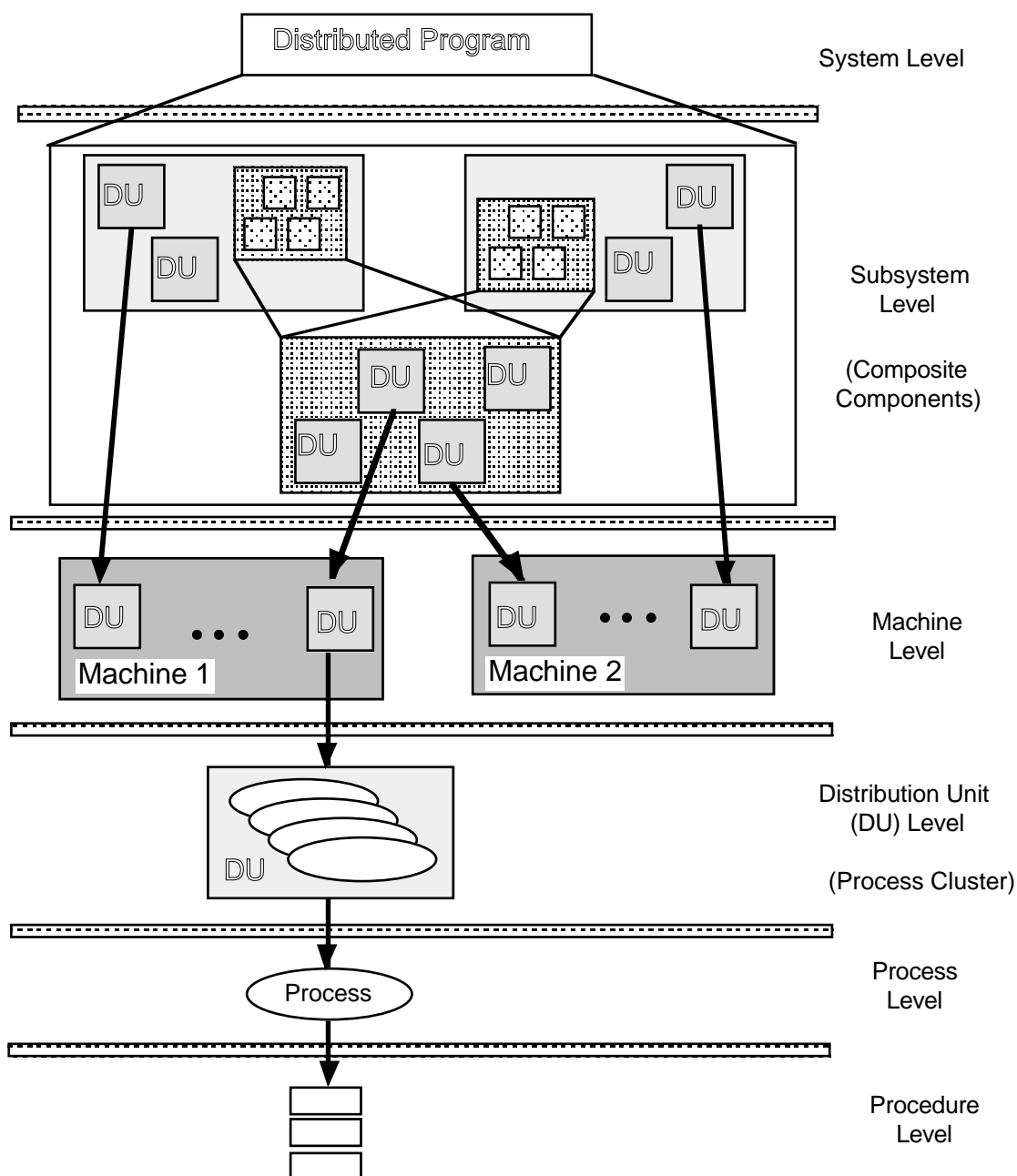


Figure 5.6 Hierarchical System Structure

At the procedure level we may wish to know the number of invocations of a particular procedure or the exact times at which it was called and when it returned (c.f. profiling in UNIX). At a process level we are interested in process concurrency, interactions and interdependence, e.g., the number and perhaps contents of messages sent and received by a process. At machine level, CPU usage (active, blocked, ready and idle times) may be

significant, whereas at system and subsystem level it may be necessary to observe the logical interconnection of distribution units. A Distribution Unit, is often called a cluster and would typically be a set of lightweight processes or threads in an address space. A subsystem can be represented by a composite component which defines internal component instances and their interconnections.

It is important for the user to be able to focus on the monitoring information that is of immediate relevance without having extraneous information cluttering the display. This can be achieved by focusing on a particular level and using the combining and filtering techniques, discussed previously, so that the behaviour of a distributed system can be viewed at a desired level of abstraction and detail.

In TMP [Haban & Wybraniec 90], a menu-driven graphical display has been provided which presents system behaviour with regard to the hierarchical system structure. With the aid of the structure and type information stored in a *program graph*, a central station is able to graphically display the logical structure of the distributed system on the screen (figure 5.7). The user can start from the highest abstraction level, and refine her point-of-view by interactively zooming through the hierarchical program structure. At each level performance metrics are presented in suitable, easy-to-read charts and graphs. The volume of communication between different subsystems are visualised by the width or colour of the lines representing the interconnection between the modules.

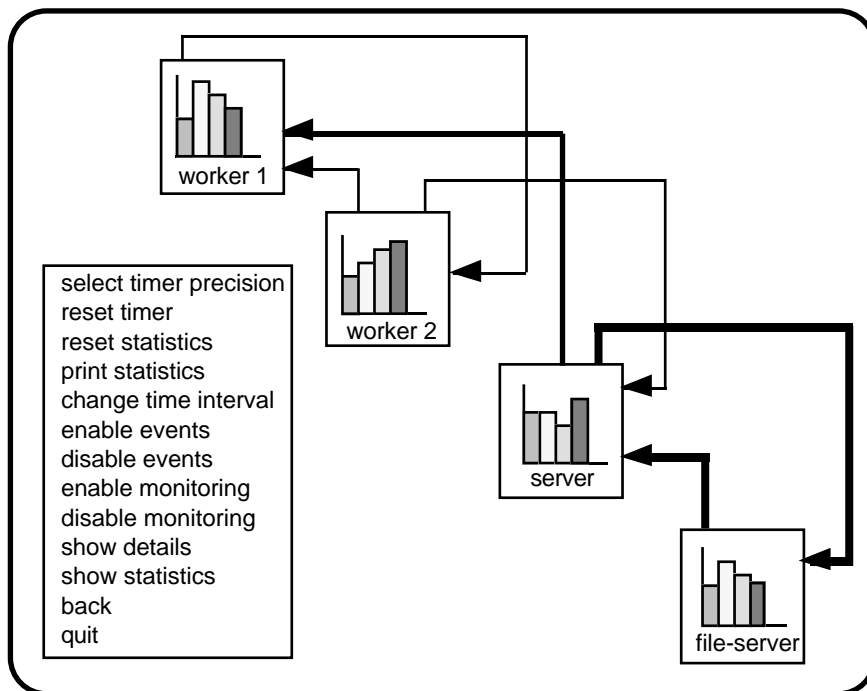


Figure 5.7 Display of the logical system structure

A further display format allows the user to focus on interactions between program units without following the hierarchy imposed by the programmer (e.g., observing the interactions between two processes belonging to two different process clusters).

ConicDraw [Kramer et al. 89] is a graphical tool which can maintain representations of executing Conic systems in terms of software component instances interconnections, and execution state. It supports on-line monitoring of systems and provides graphical and textual presentation of a system's configuration state. Clicking on a selected composite component opens it up to display the configuration of its internal components. This enables the user to navigate up and down the component hierarchy and view the system at different levels of abstraction.

Observation of system activities need not be restricted to exactly one level of abstraction at a time, and it would be useful to enable the user to observe an activity at several levels simultaneously.

ii) 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 comprehension of potentially very cluttered display.

In time process diagram of IDD [Harter et al. 85] the user can move the rows so that information about the related processes can be placed close together.

ConicDraw [Kramer et al. 89] provides various facilities expected of a diagram editor. It allows the user 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.

iii) Controlling the Time of Display

Some display tools provide a history function, permitting the user to scroll the display forward or backward in time, and control the speed at which the behaviour of the system is observed, as in SIMPLE [Mohr 91] and Radar [LeBlanc & Robbins 1985]. This is done by providing the following display options:

- Start and stop,
- Interrupt,
- Restart,
- Step-by-step display,
- Continuous display (real time or slow-motion)

iv) Use of Multiple Views

As mentioned before, one single view of system activity may not be enough and multiple complementary views may be needed to enable the user to obtain a more comprehensive picture of system behaviour. This can be achieved by using multiple windows presenting the system activities from different points of view.

Voyeur [Socha et al. 1989] is a prototype system that facilitates the construction of multiple application-specific, visual views of parallel programs using language-independent and system-independent mechanisms. It concentrates on views that are close to the programmer's mental model of the problem.

v) Visibility of Interactions

Some tools enable the user to display the contents of a particular message (e.g., the Radar [LeBlanc & Robbins 1985]). More specifically, it would be useful to be able to chose an event or status report and display its contents (e.g., its time stamp, object identifier, etc.). The width or colour of the lines representing the components' interconnection can represent the volume of communication between them (e.g., TMP [Haban & Wybranietz 90]).

6 IMPLEMENTATION ISSUES

6.1 Intrusiveness of Monitoring Systems

Intrusiveness is the effect that monitoring may have on the behaviour of the monitored system, and results from the monitoring system sharing resources with the observed system (e.g., processing power, communication channels, storage space). 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 execution time of the application,
- masking or creating deadlock situations.

This means that the results of monitoring with an intrusive monitor can only be taken as an approximation of what happen in an unmonitored system [Lummp et al. 90].

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 categorised into three types: hardware monitors, software monitors and hybrid monitors.

6.1.1 Hardware Monitors

In this class of monitoring systems a separate object (a hardware monitor) is used to detect events associated with an object or group of objects. It performs the detection by observation of system buses or using physical probes connected to the processors, memory ports, or I/O channels, [Marinescu et al. 90, Tsai et al. 90].

It has the advantage of being nonintrusive. This is achieved by separating the resources used by the monitoring system from those used by the monitored system, so that the monitoring system has minimal 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 lot of monitoring information has to be generated and processed very rapidly. The disadvantages are that:

- They require additional hardware, and therefore are more expensive.
- Generally, they provide very low-level data and do not meet the requirements of application programmers in parallel environments. Usually considerable processing and complicated mechanisms are required to provide application level monitoring information, from low machine level data.
- Hardware monitors form the least portable class of monitoring mechanisms. Their installation requires great expertise and thorough knowledge of the system, as they often use specific and sophisticated features of the hardware [Wybranietz & Haban 90]. Nowadays, the design of hardware monitors are greatly complicated by the use of pipelining and on-chip cache to increase the throughput of microprocessors and also an increase in the integration of various functional units (e.g., floating point units and memory management units) which makes monitoring difficult. In the future, this will lead to integration of monitors on the chip [Bemmerl et al. 90].

6.1.2 Software Monitors

Software monitors usually share the necessary resources with the monitored system. The program is instrumented by inserting *software probes* in the code to detect events. Use of software monitors has the following advantages:

- Monitoring information is presented in an application-oriented manner which is easy to understand and use, compared to low-level information generated by hardware monitors.
- Software monitors are portable and can easily be replicated.
- Compared with hardware monitors they are easier to design and construct and are more flexible.
- No additional and dedicated hardware resources are required. This makes them much cheaper 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 behaviour. 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. Jade [Joyce et al. 87], and Meta [Marzullo et al. 91] are examples of systems which use software monitors (see section 7.2).

There are various approaches to software instrumentation [Van Riek & Tourancheau 91]:

i) Instrumenting the Source-code

Software probes are inserted into the source-code of the program at appropriate points. It provides a flexible, portable and powerful monitoring facility. The disadvantage is that modification of the probes requires the recompilation of affected parts of the code.

These probes may be inserted manually or automatically. An example of manual instrumentation of the source-code is in Meta [Marzullo et al. 91], where the programmer instruments the application and its environment with *sensors* and *actuators*. Obviously manual instrumentation is hard, time-consuming and prone to error. A monitoring system, in which the programmer uses an event specification language and includes a monitoring segment with the program, is described in [Lumpp et al. 90]. The monitoring segment is used by the compiler to automatically insert the necessary probes in the source-code. Instrumentation of complex parallel or distributed systems is a too complex a task to be done intuitively [Hofmann, et al. 92]. In the SIMPLE environment [Klar et al. 92] a tool for Automatic Instrumentation of C Object Software (AICOS) is available for instrumentation of procedures, procedure calls or arbitrary statements written in the programming language C (see section 7.1).

ii) Instrumenting Library Routines

As an alternative to the previous method, software probes can be inserted in the source-code of library routines. The behaviour of the program can be monitored by calling the instrumented library primitives. Events can be detected and reported by the probes when the instrumented libraries are used. One advantage of this method is that it provides a high-level of portability. Any program which uses the instrumented library primitives can be monitored directly. Another advantage is that when the monitoring is no longer required, the program can be linked with an un-instrumented version of library rather than a complete recompilation being needed. The disadvantage of this method is that only those events for which software probes are inserted in the library routines can be detected and reported. To overcome this problem a special trace-function can be provided and manually inserted by the application programmer to detect a particular event.

Jade [Joyce et al. 87] is an example of a monitoring system which uses instrumented library routines. In order to detect interprocess communication (IPC) events, processes are loaded with a monitorable version of the IPC protocol.

iii) Instrumenting the Object-code

Software probes are inserted in the object-code or in an intermediate representation of the program at compile time [Malony et al. 89]. This is done by using a special instrumenting compiler. Instrumenting an intermediate representation rather than the object code has the advantage that instrumentation is machine-independent and can be transparent to the programmer. It results in less overhead than source instrumentation, because low-level machine instructions can be used rather than high-level source statements that need to be compiled. The disadvantage of this method is that it requires a special instrumenting compiler.

iv) Instrumenting the Kernel

Software probes are inserted into the code of the kernel to detect system events. These software probes are executed when an application program calls a kernel function (e.g., message send or receive). It is similar to object-code instrumentation technique and has the advantage of making event detection transparent to the application program. One disadvantage of kernel instrumentation is that only events related to kernel calls can be detected and not application events (e.g., changing of the value of an internal variable in a process). To overcome this problem, a special trace-function can be added to the kernel which can be called when required. An example of kernel instrumentation approach is instrumentation of MMK operating system kernel in TOPSYS [Bemmerl et al. 91].

6.1.3 Hybrid Monitors

Hybrid monitors are designed to benefit from the advantages of both hardware and software monitors, while overcoming their inefficiencies. They have their own independent resources but also share some of the resources with the monitored system. Typically such a system consists 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 prefer the hybrid approach. An example of that is Test and Measurement Processor (TMP) which allows measuring, monitoring, testing and debugging of distributed applications [Wybranietz & Haban 90]. The designers of TMP claim that the degradation in the performance of the monitored system is less than 0.1%. TOPSYS environment [Bemmerl et al. 91] supports software, hardware and hybrid monitoring. ZM4 [Hofmann et al. 92] supports both hybrid and hardware monitoring (see section 7.1).

6.2 Global State, Time and Ordering of Events

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

These features can affect both the behaviour 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 behaviour of the system non-deterministic and

unpredictable. Furthermore, arbitrary message delays makes it impossible to obtain an instantaneous and consistent "snapshot" view of the system. Also, 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 analysing monitoring traces.

We shall briefly describe a number of approaches which are used to overcome this problem.

i) Physical clock synchronisation

The aim is to obtain a unique physical time frame within a system, where each processor maintains its own local physical clock. Physical clock synchronisation is based on the exchange of messages containing time-stamps. These may contain an external time stamp received from an accurate radio time signal or local time stamps and the nodes try to maintain processor clocks within some maximum deviation of each other.

For example [Lamport 78] proposed an algorithm which assumes that both a lower bound (min) and an upper bound (max) are known for message delays. Every T seconds each process sends a **synch** message on all of its channels, to other processes. This protocol belongs to a deterministic class of protocols which cannot guarantee a precision better than $(\max - \min)(1 - 1/n)$, where n is the number of clocks which have to be synchronised. Deterministic protocols offer a high probability of successful synchronisation with small number of messages, at the expense of low precision.

[Christian 89] proposed a probabilistic approach for reading remote clocks subject to unbounded message delays, which offers a higher precision than the best achieved by the deterministic protocols, but which carries with it a certain risk of not achieving synchronisation.

One disadvantage of keeping clocks synchronised is the additional overhead that it introduces into the monitored system. [Duda et al 87] described an off-line approach to this problem which is to record local traces of external events, using (unsynchronised) local clocks and to analyse them after the execution of a distributed program to deduce global properties or global performance indices. The global time is estimated from local traces with a desired precision. A least-square regression analysis is used to estimate the time offset and the time offset rate between two local clocks.

ii) Logical clocks

A system of logical clocks, based on the *causality relation*, can be used to establish a partial ordering of events in the system [Lamport 78]. The system of logical clocks can be represented by a function LC which assigns to any event e a locally maintained timestamp LC(e), such that the following condition is satisfied:

For any events a and b, if $a \rightarrow b \Rightarrow LC(a) < LC(b)$

where $a \rightarrow b$ means a precedes b

To satisfy this condition each process must increment its clock between any two events. Also upon receiving a message, a process must advance its clock to be later than the time stamp of the message.

This scheme has several advantages. It imposes small overheads on the monitored system. The quantity of information contained in messages and maintained by each process is minimal – an integer. The system can easily accommodate dynamic changes in that a component can be added to the monitoring domain with no changes to the ordering scheme or other components.

A problem with the logical clock approach is that it lacks what is called *density*. Given e_1 and e_2 where $LC(e_1) < LC(e_2)$, it is not possible to determine if there is another event e_3 such that $LC(e_1) < LC(e_3) < LC(e_2)$. This is particularly important for run-time re-ordering of events where

we want to detect message loss or delays. Another problem with both logical and physical time is that although they are consistent with causality, they do not characterise it [Schwarz & Mattern 92]. It is not always possible to determine whether the two events are causally related or concurrent, by looking at their timestamps.

iii) Vector Clocks

[Fidge 88] has proposed an approach based on a *vector of logical timestamps*. Rather than one clock value, each process P_i maintains a vector of logical timestamps VC_i , where every element of the vector corresponds to one interacting process. Intuitively, element j of P_i 's logical vector for event e is the number of events that P_i knows P_j has executed upto e . This vector is maintained by a process and included in outgoing messages. The vector is updated whenever a message is received from a process. Although vector clocks can overcome the problems outlined above the overheads could be very high. A logical clock has to be maintained by every process for all other processes with which it communicates. The vectors of timestamps included in messages can be quite long, thus increasing the communication overhead. Also dynamic changes to the monitoring domain becomes much more difficult.

iv) Global Snapshots

In contrast to the approaches described above [Chandy & Lamport 85] proposed a technique which concentrates on constructing a global snapshot of system activity. This technique encompasses the entire system in the gathering and grouping of state information. This snapshot is constructed through the transmission of *marker messages* over every communication link in the system (FIFO channels are assumed). Any process can initiate such a snapshot by saving its local state and transmitting marker messages over each of its outgoing links. Upon receiving its first marker, a node is incorporated into the snapshot, saves its local state, begins recording messages on its incoming links and transmits a marker over each of its outgoing links. A node stops recording the information received on a link when a marker is received over it. The dissemination of these marker messages throughout the system serves to create a timeslice, which demarcates the edge of a snapshot. The global snapshot can then be calculated from all of the local states (snapshots) of the nodes and all of the channel information recorded. Partial ordering is not violated by the system view captured.

The major disadvantage of this technique is that all the processes and communication links are involved in the formation of a snapshot. This is highly inefficient means of gathering state information to recognise an event whose scope was limited to only a small subset of the system processors [Spezialetti & Kearns 89].

For general monitoring purposes, in addition to the ordering of events, we are interested in the time intervals between occurrences of different events. For this purpose, using a combination of logical vector clocks and synchronised clocks may be preferable. Obviously the overheads involved would have to be considered.

7 SOME EXISTING MONITORING SYSTEMS

In this section we describe four of the existing monitoring systems in more detail.

7.1 ZM4/SIMPLE

A monitoring system which allows the observation and analysis of the functional behaviour and the performance of programs in distributed systems is described in [Hofmann et al. 92, Mohr 90, 91]. This monitoring system is used for performance evaluation, tuning and debugging. Their approach is based on what they call model-driven monitoring.

7.1.1 Model-driven Monitoring

In model-driven monitoring, event-driven monitoring and event-based modelling are integrated into one methodology because they both rely on the same abstraction of the dynamic behaviour: the event.

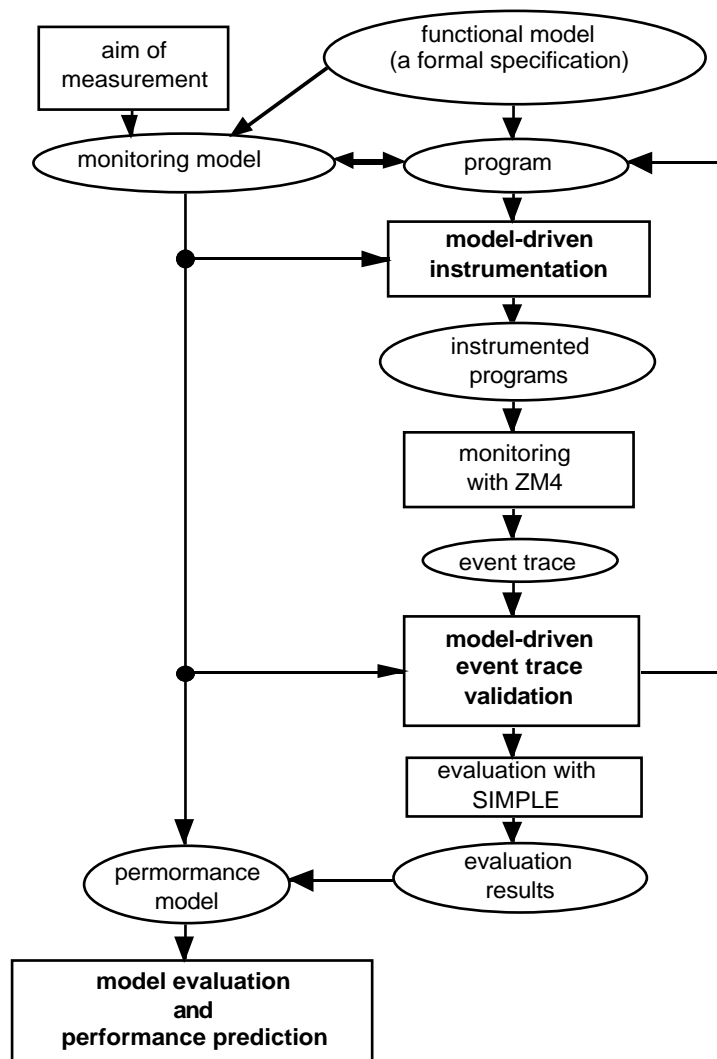


Figure 7.1 Model-driven Monitoring

An event-based formal model of the program behaviour using graphs, petri-nets, or queuing models is used to derive a monitoring model³. The monitoring model is a subset of the

³Model-driven monitoring is independent of the modelling method.

functional model which uses the same set of events to describe the behaviour of the program at a desired level of abstraction. It is then used, with the help of certain tools, for systematic program instrumentation (model driven instrumentation), event recognition, event trace validation, and for creating a performance model as shown in figure 7.1. Model-driven instrumentation guarantees a one-to-one mapping between the model events and the monitoring events. The tool AICOS has been developed to allow Automatic Instrumentation of C Object Software. Arbitrary statements in the program under investigation can be instrumented. The instrumented program is executed and it generates an event trace. The monitoring model is systematically checked against this linearly ordered trace and program errors are detected during the model-driven validation stage. If the trace is valid, it is evaluated.

The monitoring model is transformed into a performance model by adding timing and frequency attributes derived from a measured event trace. It is used for validating the dynamic behaviour of the program and can be used for automatic event trace validation and performance prediction of not yet available systems and implementations.

7.1.2 The ZM4/SIMPLE Monitoring Environment

i) ZM4

Instrumented objects write event tokens to the hardware interface of ZM4 (figure 7.2), which is a distributed hybrid monitoring system. It is structured as a master / slave system with a *control and evaluation computer* (CEC) as the master and an arbitrary number of *monitor agents* (MA) as slaves. The master controls the measurement activities of the MAs, stores the measured data and supports the user with a powerful toolset for evaluation of the measured data. Each MA has up to 4 *dedicated probe units* (DPUs) which are printed circuit boards and link it to the nodes of the object system. The MAs control the DPUs and buffer the measured event traces on their local disk. Event traces are transferred to the CEC for evaluation. The DPUs are responsible for event recognition, time stamping, event recording, and for high-speed buffering of event traces. Event recording is independent of the object system.

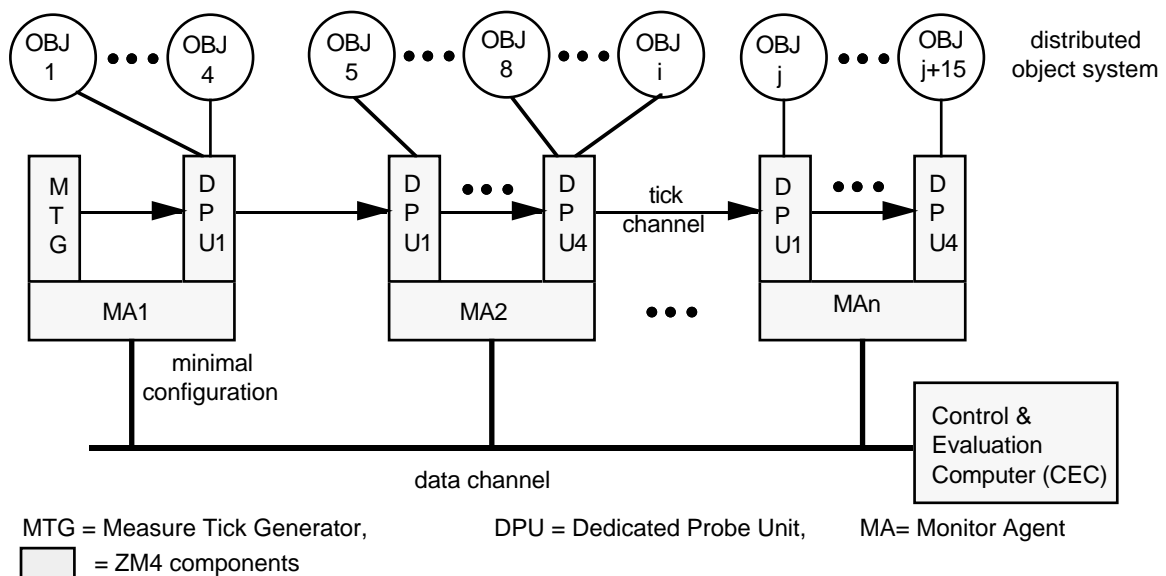


Figure 7.2 Distributed architecture of the ZM4

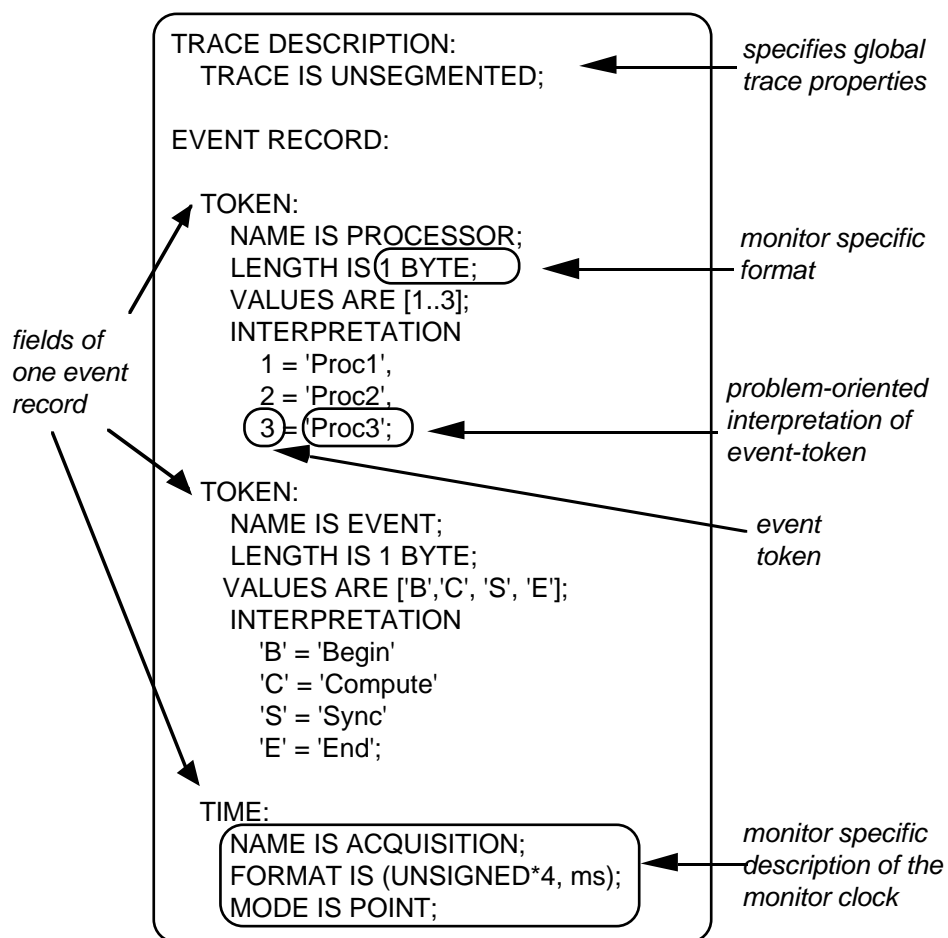
A local clock with a resolution of 100 ns. and a time stamping mechanism are integrated into the DPU. The *tick channel* is used to synchronise the local clocks of the DPUs to the master clock on the *measure tick generator* (MTG). The data channel (Ethernet with TCP/IP) forms the communication subsystem of the ZM4 and it is used to disseminate control information and measured data.

ZM4 is scalable and large object systems are matched by more DPUs and MAs, respectively. ZM4 can record events of arbitrary objects with arbitrary physical event representation and format.

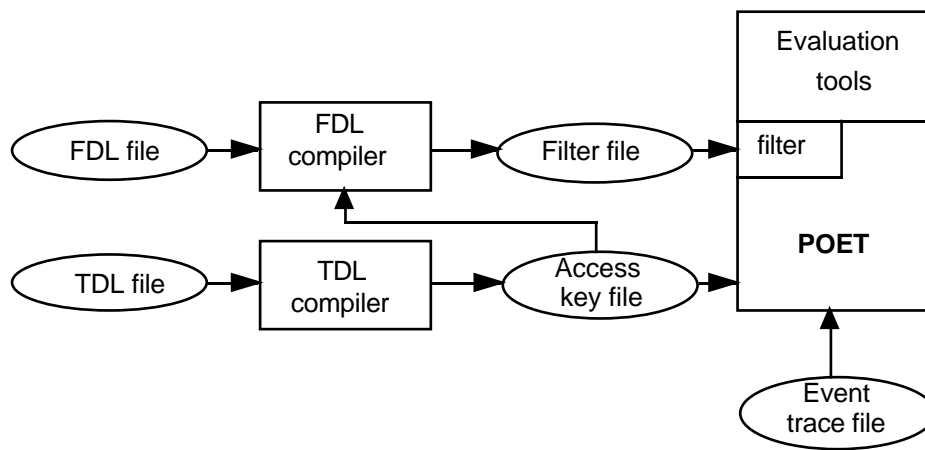
ii) SIMPLE

SIMPLE is a tool environment designed and implemented for performance evaluation of arbitrarily formatted event traces and runs on UNIX and MS-DOS systems. The measured data is considered as a generic abstract data structure or an object, which can only be accessed via a standard set of generic procedures. This enables the user of such reports to abstract away from different data formats, structures, representations and meanings and thus become independent of monitored system and monitor devices used. Events can be identified by application-oriented names to give reference to the source.

The formats, structures and properties of monitored data are described in a trace description language (TDL) which clearly reflects the fundamental structure of an event trace (see figure 7.3).



7.3 Event trace description in TDL



7.4 Event trace access with TDL/POET/FDL

The TDL description is compiled and an access key file is generated (figure 7.4). This file is used by a standardised Problem Oriented Interface (POET). POET enables the user or evaluation tools to access monitored data stored in event trace files in a user-defined, problem-oriented manner. A language called Filter Description Language (FDL), similar to TDL, has also been developed to specify rules for filtering event reports depending on their contents. The problem-oriented identifiers of TDL file are also used for filtering. By using TDL/POET/FDL all tools of SIMPLE are independent of the properties of a monitored system, especially its operating system and programming languages.

iii) Processing of Monitoring Information in SIMPLE

The tools available for processing of monitored information include:

- *MERGE*: takes the local event trace files and the corresponding access key files as input and generates a global event trace and the corresponding access key.
- *CHECK TRACE*: performs simple validation and plausibility tests on an event trace, e.g., checking the validity of the time stamp, or whether the token fields contain only defined token values.
- *VARUS*: performs more detailed, application specific, validation checks specified as assertions in a formal language. For example:
 - i)

```
ASSERT ( EVENT=='Compute' AND PROCESSOR=='Proc1') ALTERNATING
WITH   ( EVENT=='Sync'      AND PROCESSOR=='Proc1')
ELSE   "sequence error on processor 1";
```
 - ii)

```
ASSERT NUMBER (EVENT=='Begin')==NUMBER (EVENT=='End')
```
 Rule (i) states that the Compute and Sync events on processor Proc1 should alternate. Rule (ii) specifies that the number of Begin and End events should be the same.

Both CHECK TRACE & VARUS generate a report containing all errors detected.

- *Filter*: allows the user to select event records depending on their record fields.
- *ADAR* (Activity Definition and Recognition System): enables the user to combine lower level event to form higher-level events called activities, and assign new attributes to them.
- *TRCSTAT*: performs simple statistical computations on an event trace. It computes frequencies, durations, and other performance indices. For more complex computations the data analysis package S from AT&T is used. It provides a high-level programming language for data manipulation and graphics.

iv) Presentation of Monitoring Information in SIMPLE

- *LIST*: generates a simple textual list of events and permits the specification of which event record fields are to be printed and their format.

environment. A sensor can be polled, at intervals defined by the programmer, to obtain its current value or a *watch* can be set up that alerts the client when the sensor value satisfies some predicate. Built-in sensors obtain information directly from the run-time environment e.g., CPU and memory utilisation. Meta provides the *read-var sensor* for reading the values of certain kinds of global variables in an active process. There are also user-defined sensors which are implemented by the programmer and registered with Meta at run-time e.g., application throughput, or a queue length.

Built-in actuators can be used to change a process's priority, a global variable (using *write-var* actuator) and user-defined actuators can be used to change the application's behaviour.

7.2.2 Structure Description

The programmer describes the structure of the application using Lomita's object-oriented data modelling facilities. Meta provides an object-oriented temporal database in which the application and environment provide the data values. Components in the application and the environment are modelled by entities, following entity-relationship database terminology. The resulting model is used by the control program at the Policy Layer.

To provide higher level views of the behaviour of the program Meta allows the combination of multiple sensors, or values of one sensor over a period of time in the form of *derived sensors*. Lomita provides simple arithmetic operations, and also functions for min, max, size and median etc. that operate over sets of values e.g.

sensor load_ratio: **real** := SigPro.load / Machine.load

sensor high_load: **integer** := max(history(load, 600))

where load_ratio is the derived sensor calculated from primitive sensors SigPro.load and Machine.load and high_load is a derived sensor representing the maximum load over the last 10 minutes (600 seconds)¹.

7.2.3 Expressing Policy Rules

Using the data model, the programmer defines of a set of Lomita policy rules which describes the intended behaviour of the system and can make direct calls on sensors and actuators and other functions in the data model. e.g. **when** condition **do** action

where condition is a predicate expressed on the underlying data model and the action component is a sequence of actuator invocations and data model operations (e.g., create or delete). The condition may contain interval temporal logic expressions, in which case they are converted to finite state automata. The following temporal operators are used in Lomita:

- During I always P:** true if and only if predicate P is true throughout time interval I.
- During I occurs P:** true if and only if predicate P is true at some point within time interval I.
- P until Q:** Expresses the time interval beginning when predicate P next becomes true, until predicate Q subsequently becomes true.
- P for T:** Expresses the time interval beginning when predicate P next becomes true, until T seconds have passed.

7.3 Demon

Demon [Demon 93] is primarily a flexible visualisation tool (based on UNIXTM and X Window SystemTM) which allows observation of the behaviour of a (possibly distributed) system at different levels of abstraction. It is a *centralised* tool that could receive messages,

from various sources such as a live network, or previously recorded files and could perform various monitoring functionalities such as filtering, correlation, analysis and presentation.

To perform these functionalities, Demon follows a set of rules written in the *Demon rules language* (also called *configuration language*) which is similar to a traditional procedural language. These rules are contained in a *rules file* which is compiled by a special rules processor. They control the three main stages of Demon's operation: *recognition*, *interpretation* and *depiction*. At recognition stage incoming messages are used to detect events. At interpretation stage, these events are used to update an internal model of the monitored system or send out messages to various places. The selected parts of the internal model are displayed at depiction stage.

Demon has no knowledge of the distribution of the system under scrutiny. It merely traps messages (i.e., monitoring reports) entered manually by the user in a file, or those generated by a live network through the use of specially instrumented communication routines. Certain routines have been provided to handle a range of communication protocols.

Demon expects messages in a predefined format, each message carrying with it type information of its data fields. To collect these messages from a network, a separate (external) network handling process must be used which is connected to Demon via a pipe. This external process communicates with the monitored system using TCP/IP protocol.

Demon can save the incoming messages in a specified file, for later analysis (i.e., trace generation). Various options are provided for the user to select a new file or open, close or append to a specified file. Recording is a rather simplistic operation. It is performed before any filtering can take place (i.e., record **all** the messages that are coming in!) and only one file can be specified for recording. A more sophisticated recording mechanism which incorporates its own filtering and other capabilities such as merging and trace splitting, would be more useful.

A recording file may be created from customised messages entered manually by a user in a text file and processed by special message handling utilities. Therefore messages may be injected and Demon can operate stand-alone. This is particularly important for simulation and demonstration purposes. Manually entered messages must start with the number of fields followed by a list of (field type, value) pairs. The field types may be unsigned long, long, bool, char, float and strings. The stored messages can be processed automatically at a preset rate or stepped through manually.

7.3.1 Event Recognition

An *event* is recognised by the Demon when the contents of an incoming message satisfy certain predicate, as defined in the rules file. An explicit filtering is performed here, where messages which do not satisfy those predicated are discarded.

7.3.2 Interpretation

This stage includes two important activities. Updating the internal model and sending out messages to various destinations such as the monitored system or another system. The first functionality corresponds to the database updating stage in our model. The second could correspond to the dissemination stage in our model and control activities which are not considered in the model.

(i) Updating the Internal Model

The internal model is a directed graph whose basic components are nodes and arcs. The nodes and arcs can represent anything and are provided as a basic template to which any meaning can be attached. For example, nodes could be components on a network and arcs could be messages between them. Various attributes and parameters can be associated with nodes and arcs.

Information about the events can be used to update Demon's internal model to reflect the changes or send out messages (i.e. active demon). This is entirely under the control of the user. Events could cause arcs and nodes to be created, examined, modified or destroyed.

The nodes in the internal model are hierarchical, with parent-child relationships. A node may have a number of children and a single parent node. The topmost node is called the ROOT. This hierarchical structure permits a wide range of views of system to be constructed.

(ii) Active Demon

Demon can send string-type messages to the monitored system, to another system, to a file or any other UNIX output device. These messages may be sent automatically in response to events, or they can be initiated by the user (e.g., provided as an option in a pop-up menu associated with a displayed node).

7.3.3 Graphical Presentation

Demon allows users to define their own graphical presentations, in a flexible manner. It has facilities for customising and implementing iconised presentations of the behaviour of the system.

Every node (icon) displayed on the screen has a number of attributes and a pop-up menu associated with it, which can be displayed using the mouse. An *In* option on this menu can be used to zoom in on the node. An *Info* option allows the user to see the general attributes, and any explanation texts associated with it. Also, various options associated with active demon can be provided so messages can be sent under the control of the user.

For an arc similar options could exist but they are associated with its label, if it has one. Therefore, if an arc represents a communication link, messages on that link can be displayed, in addition to other relevant information.

A *view* shows part of the information contained in the internal model. The configuration language allows the user to create many different and simultaneous views of the system, each shown in a separate window. For each view the user must define:

- the *section* of the internal model to consider
- the *conditions* under which this section is shown
- the *appearance* of the nodes and arcs when they are drawn (possibly depending on certain conditions)

The section is defined by two parameters - a node position in the model, and a number of levels down. The conditions for display may depend on many factors such as message contents, global variables, etc. The appearance of the nodes and arcs is flexible and various colours and shapes, including bitmaps, may be specified.

The user can select one or more of the available views. This is the second stage of filtering. Within one view in a window various options (buttons) are provided:

- *Views*: used to switch to alternative views
- *Zoom-out*: to zoom out one level
- *Magnify*: to increase the size by 10%
- *Shrink*: to decrease the size by 10%
- *kill*: to close the current view window

Also a button called *visual step* is provided, which allows the user to step through the messages until one affects the current view.

A Flow Control Panel is provided to allow the user to control the way Demon steps through messages it receives. Options provided on this panel are:

- *Recognised step*: When this button is pressed, Demon reads and processes the messages until one causes an event. (This may or may not affect the current views.)
- *Visual step*: Messages are read and processed until one affects the current views.
- *Play*: Demon automatically steps through the messages. A Delay Control slider has been provided to control the delay between two steps. The delay could be from hundredths of a second to a maximum of (about) one second.
- *Stop*: Stops Demon automatically stepping through the messages.
- *Clear*: This button destroys the internal model and clears the current views.
- *Reset*: Demon is reinitialised. It includes resetting the source file to the beginning, destroying the internal model and clearing the screen.

A rewind option would have been useful to re-display the same sequence without having to reset and start the whole thing again. Such an option is not provided.

7.3.4 Programming Demon

The rules program describe how Demon is to recognise events, how it is to interpret them and how it is to present its model of the system. It consists of four sections: *declaration*, *event definitions*, *action definitions*, *view definitions*.

(i) Declaration section

The declaration section starts with the keyword DECLARATIONS. In this section the following items are declared: message fields, node and arc types, constants, types, variables, tables and sets. Apart from the first two items, the rest are optional.

The message declaration section starts with the keyword MESSAGES. A message field declaration consists of a *name* which is used to refer to the data field, a *location* which is the position of the field in the Demon's standard message, and the *type* of the data field.

e.g., ObjectID PARTITION 0 SELECTOR 1 TYPE WORD;

A node or arc is declared by using the keywords NODE or ARC, a name, and the details of its description fields (i.e., its attributes). The description fields of a node or an arc could contain anything.

e.g., NODE Object (STRING Name, Type; LONG Desig, CreateTime);

Constant, type and variable declarations are similar to those of other procedural languages.

A *table* provides a simple database structure. It stores a value against a key. You are free to store any type of data. The table declaration section starts with the keyword TABLES. A table is declared by giving its name, the type of its key and the type of the value stored against the key.

e.g., Designator LONG NODEID;

A set is an unordered collection of items, stored for future reference. The set declaration section starts with the keyword SETS. Each set declaration consists of a name and the type of its elements.

e.g., Loads LONG;

Some useful set-oriented operators seem to be missing from the language (e.g., max, min, average, etc.)

(ii) Event Definition

The event definition section begins with the keyword EVENTS. Events are defined based on messages that are received. Each event definition consists of an event name followed by the condition. A condition refers to message fields and a combination of conditions may be specified using various logical and relational operators.

e.g., CREATEOBJ: MessageType = 'c' AND ObjectType!= 2 OR Field3 <6;

A special predefined initialisation event (INITIAL) is generated as soon as Demon has read the rules file, and used for setting up initial conditions before any messages arrive. Any actions may be defined for this event.

Although it is claimed that the user may refer to other events when defining a particular event, this feature is not presented in the documentation. No temporal operators seem to have been provided (e.g., e1 : e2 -> e3).

Currently in event definitions no reference can be made to the information stored in the internal model of the system maintained by Demon. However, at presentation stage references can be made to the internal model when specifying the conditions under which a particular section of the model is to be displayed. It would be better to allow it at event recognition stage as well to provide earlier filtering.

A defined event is activated by default. It would be better to separate event definition and activation. This will allow dynamic filtering of events, while presenting the user with a set of potential events that may be reported.

(iii) Action Definition

For each defined event a set of actions may be specified, with the following format:

```
ACTION      event : <actions>      ENDACTION
```

When *event* occurs the sequence of *actions* is performed. This part of the program is similar to a conventional procedural language program. Usual flow control operations, basic mathematical operations, and operations to manipulate the internal model, tables and sets are available. Also using a SEND function it can send messages to the monitored system, to another system, to a file, or to any UNIX output device.

(iv) Defining Views

This is the final section of the Demon rules program. It describes how to build the various views of system behaviour. Each view is defined by a template, which specifies:

- which part of the internal model to consider for displaying
- under what conditions those nodes and arcs will be drawn
- what is their appearance (e.g., colour, shape, scale)

- which template is adopted on zooming out a level
- which other templates are to be available to change to from the current view

The view definition section starts by the keyword VIEW followed by the region definition (if any), and then all the view templates. Facilities are provided to incorporate bitmaps in X bitmap format. Grids can be defined for automatically positioning nodes on the screen.

7.3.5 Start-up Options and Operations

Various start-up options can be defined from the control line or from a resource file. They specify the rules files, source files, output channels and display lists to be used by Demon. Once Demon has been started, operations are controlled through a Motif-based GUI. This presents a main control panel incorporating iconised buttons to operate and to modify Demon. These select:

- *source* Where Demon's messages are coming from
- *output* Where the "Active Demon" messages are sent to
- *Rules* Which rules to use in processing the messages
- *Views* Which views of the internal model are to be displayed
- *Flow* How demon will process the messages (step by step, continuous, etc.)
- *Record* Where network messages are to be stored

Pressing these icons brings up appropriate dialogue boxes. With source and output options, it is possible to view the messages as they are received or sent, but it seems that only one source and destination may be selected at any time. An improvement would be to allow Demon to receive messages from multiple sources or send them to several output destinations simultaneously (e.g., to monitor two separate networks at the same time). A similar criticism applies to recording option, where only one file can be specified and **all** the incoming messages are stored in that file with no filtering.

7.3.6 Conclusions

Demon provides a lot of flexibility in the way various monitoring functionalities are specified, particularly with respect to visualisation. There are however a number of major criticism that makes it unsuitable as a tool for monitoring large and complex distributed systems.

Demon is essentially a *centralised* tool, and therefore as far as monitoring large and complex distributed systems is concerned, it is not scalable. In fact it has no knowledge of the distribution of the system under scrutiny and therefore important issues related to distribution such as ordering of events, consistency of the global state, validation of monitoring data, instrumentation, activation and intrusiveness are not addressed.

Another major criticism of Demon is that various monitoring functionalities (e.g., event recognition, interpretation or correlation and presentation) can only be specified statically. These activities are specified in a fairly low-level language, in the rules file, which has to be compiled before it is used. For managing large, complex and evolving distributed systems, monitoring domains and requirements may change frequently and therefore the managers should be able to specify these activities in a dynamic fashion.

To specify events no reference can be made to the information maintained in the internal model and events are defined using only the contents of the messages that are received. No temporal operators has been provided and therefore using the current language, temporal specification of events is either not possible or difficult. Also, event definition and activation have been combined. A better policy is to separate these to allow dynamic activation / deactivation of

events. This makes dynamic and implicit filtering possible, which further reduces intrusion and overheads.

With respect to "Active demon" issues such as consistency and atomicity of actions are not considered. For dissemination of monitoring information to a number of managers more powerful mechanisms are required (e.g., a subscription and dissemination mechanism).

Rather than the static specification of graphical presentations, a graphical editor may be used to allow users to create and store their own views in a dynamic fashion. Also, a set of predefined graphical representations (e.g., icons, lines, etc.) could be provided as options in a pop-down menu for the user to select using a mouse.

7.4 Monitoring Databases

A data-oriented approach to network management in which monitoring and control are specified as data manipulation statements on a network database, is presented in [Wolfson et al. 91]. In their approach a conceptual data-base model of the network is constructed and continuously updated to represent the current status of the network. Monitoring is done by "watching" for certain important events to occur. An event is represented by a data pattern, and watching for it means the continuous retrieval of this data-pattern from the database.

They have extended the SQL data manipulation language to include new features which allow real-time and temporal monitoring of database changes. Two types of data are stored in the network database. The configuration data gives the current status of the network and history data consists of trace information about the evolution of the network and its status over time.

7.4.1 Events

For a primitive or correlated event to occur it has to be specified and then activated. After the occurrence the specification has to be reactivated in order for the event to occur again. Basic events consist of:

- (i) *Data-pattern events* occur when a certain data-pattern appears in the data-base. This is equivalent to the data flow events in our model. Such an event is specified using a data-retrieval operation which is executed if only one of the retrieved objects changes. One of the parameters of such an event is PERSISTANCE. If PERSISTANCE \geq "time-interval" is associated with an event it indicates that the event is to occur only if the data-pattern persists in the data-base at least for the specified time-interval. Therefore transient events can be ignored.
- (ii) *Data manipulation events* occur when a data manipulation operation is invoked in the system (e.g., a retrieve, add, delete, replace or update). This is equivalent to the control flow events in our model. Such an event is specified by a data manipulation operation. For example, the following defines an event which occurs when a tuple is deleted from the relation LINKS which has a DELAY > 5 .

DELETE LINKS DELAY > 5

- (iii) *Calendar-time events* are specified using a date and time. e.g., "12 A.M. January 8", says the event occurs every year on January 8th at 12 A.M.

7.4.2 Trace Collection Service

The trace parameters which can be specified include trace identifier, the class of objects being monitored, the attribute whose change is being tracked, the event to activate a collection and the duration. The user can specify a selection predicate for an attribute which results in its old value being appended to the trace after a change or a separate trace for one or more objects can be requested (e.g., all the ones with COLOUR = "yellow").

7.4.3 Combination

Combined or correlated events are specified using correlation rules. For example the rule,

OVERLOAD-AT-12 :- OVERLOAD, 12 A.M.

means that the combined event OVERLOAD-AT-12 occurs if the data-pattern event OVERLOAD occurs at the same time as the calendar-time event 12 A.M. The operator “~” is used to denote negation, e.g., ~EV1 means that event EV1 did not occur. The definition of a combined event could also consist of a disjunction of two events. E.g.,

OVERLOAD-OR-12 :- OVERLOAD.

OVERLOAD-OR-12 :- 12am.

specifies that the event OVERLOAD-OR-12 will occur when OVERLOAD occurs or at 12 A.M., whichever is first.

For each rule it is also possible to specify a temporal order for events (e.g., E1->E2), and temporal constraints (e.g., {OVERLOAD, 12 A.M.} = 5 s, says that OVERLOAD-AT-12 will occur only if OVERLOAD and 12 A.M. are at most 5 s apart).

Combined events are specified using correlation rules. For example the rule,

OVERLOAD-AT-12 :- OVERLOAD, 12 A.M.

means that the combined event OVERLOAD-AT-12 occurs if the data-pattern event OVERLOAD occurs at the same time as the calendar-time event 12 A.M. The operator “~” is used to denote negation, e.g., ~EV1 means that event EV1 did not occur.

The definition of a correlated event could consist of a disjunction of two events. e.g.,

OVERLOAD-OR-12 :- OVERLOAD.

OVERLOAD-OR-12 :- 12am.

specifies that the event OVERLOAD-OR-12 will occur when OVERLOAD occurs or at 12 A.M., whichever is first.

For each rule it is also possible to specify a temporal order for events (e.g., E1->E2), and temporal constraints (e.g., {OVERLOAD, 12 A.M.} = 5 s, says that OVERLOAD-AT-12 will occur only if OVERLOAD and 12 A.M. are at most 5 s apart).

7.5 Summary of Existing Monitoring Systems

The following figure presents a brief comparison between the approaches described in sections 7.1 to 7.4, against the functionalities identified in the model.

	ZM4/ SIMPLE	META	DEMON	MONITORING DATABASES
Purpose of Monitoring	Debugging & Performance Tuning	Management of Distributed Applications	Visualisation Debugging & simulation Control	Managing Communication Systems
Status Reporting	No. Event driven monitoring used.	Yes. Polling object state. Sampling period set at instrumentation time.	No. Comms. messages are trapped.	No. Event driven monitoring used.
Event Detection & Reporting	H/w & Hybrid Monitoring. Automatic S/w instrumentation supported. Internal and external detection of process level & control flow events.	Software Monitoring. Manual instrumentation. External detection of data flow events.	Software monitoring using pre-instrumented comms. protocols. External detection by static specification of conditions on fields of any incoming messages.	S/W monitoring. Changes to data in network database detected.
Trace Generation	Yes. Trace format specified using TDL language.	No.	Yes. All incoming / outgoing messages can be stored in or appended to a specified file.	Yes. User-specified trace generation. Part of a history database.
Merging & Multiple Trace Generation	Yes. Merging supported using the tool MERGE	No.	No.	Yes. User specified merging and multiple traces.
Validation	Yes. Supported using CHECK TRACE & VARUS Tools	No. Valid event orderings guaranteed by ISIS OS.	Lamports event ordering algorithm implemented. Users may specify their own validation conditions in a rules file.	No.
Database Updating	No	Yes. A system data model is maintained and referenced.	Yes. An internal data model (a directed graph) is built & referenced.	Yes. A network database is continuously updated and referenced.
Combination	Supported using ADAR tool	Supported using LOMITA language	Supported using a language called the configuration language.	Supported using an SQL-like language
Filtering	Yes. Explicit filtering supported using FILTER tool based on event attributes. FDL language used.	Yes. Implicitly by setting polling periods at instrumentation time.	Yes. Done explicitly on incoming messages and implicitly by selecting certain display views. Specified in rules file.	Yes. Done explicitly by specifying conditions at event recognition and implicitly through activation / deactivated.
Dissemination	Monitoring information sent to a fixed central station.	Using a subscription mechanism status data is distributed to other monitoring objects.	Possible to specify multiple destinations statically, using Demon's control functionality.	Not specified
Presentation	Textual, time-process & animation display tools. User specified display rate. Graphical display.	Not specified	Flexible, user specified views and a simple replay mechanism with user controlled display rates provided.	Not specified

Figure 7.9 Comparison of Features of Example Systems

8 OSI MANAGEMENT STANDARDS

8.1 OSI Management Approach

The International Standards Organisation (ISO) have 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, but managers do not directly invoke operations on managed objects as we have assumed in figure 1.2. Instead, managers interact with a management agent which is local to the managed objects, using the Common Management Information Protocol (CMIP) as shown in figure 8.1. CMIP provides the primitives for supporting management operations to permit the managers to control remote managed objects, query state and notifications can be used by the agent to send status and event reports from the managed objects to the manager.

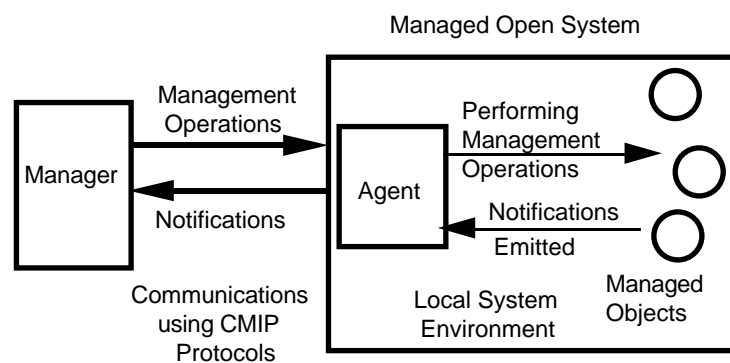


Fig. 8.1 OSI Interaction Model

Managed objects may also be used to implement elements within a monitoring system to perform processing or dissemination functions as explained below. Managed objects may be created and deleted dynamically, in response to changes in the managed system or to change the monitoring system. Managed objects constitute the Management Information Base (MIB).

The OSI standards define a set of management functions relating to configuration management, fault management, performance management, accounting and security management for communications systems, but there is no monitoring as a specific management functions. However the specifications relating to the above 5 functions and in particular fault and performance management, define elements which could be used as a generic monitoring service for distributed systems. A summary of monitoring and event reporting in OSI can be found in [LaBarre 1991]. The OSI Guidelines for the Definition of Managed Objects (GDMO) define a set of techniques and a notation for specifying managed objects [ISO 10165-4].

8.2 Generation of Monitoring Information

The state variables which are made visible by objects are called *attributes*. A generic set of states and the changes which can take place between those state are defined in [ISO 10164-2]. These states are reflected in a set of attributes which are common to all managed objects. Additional attributes may be provided to represent the applications specific status vector of the object. The typical information which would be generated in a state change event report sent to a manager includes:

- Managed object class & instance identifiers
- Event time
- Old and new values for all state attributes.
- Application specific status attributes

Objects often generate notifications to their local agent as event or status reports. The agent performs filtering and dissemination of this monitoring information using event discriminator objects as explained in section 8.3 below. An alternative, for very simple objects, is to use a *metric object* to periodically poll the managed object to read the attributes and then generate the notifications [ISO 10164-11].

The predefined event reports include

- i) *Object Management* reports are generated whenever a managed object is created /deleted, an operational / administrative state change occurs, changes occur in non-state attributes e.g., names or important operational parameters [ISO 10164-2]
- ii) *Alarm Reports* have been defined for the following alarm classes [ISO 101064-4]:
 - communication faults e.g., call set up errors, signal distortion
 - quality of service degradation e.g., problems with throughput or response time
 - processing errors e.g., buffer overflow, file access error, memory violation.
 - equipment alarms e.g., cable cut detected, locked ports or power problems
 - environment problems e.g., high/low temperature, smoke detection or excess humidity.

An alarm message can include probable cause, specific problem code, additional problem data, perceived severity, severity trend, back up status, back up object instance, threshold information, proposed repair, additional textual information actions as well as state attribute values.

8.3 Event Reporting Service

The OSI Event Reporting Service performs both the processing and dissemination functions of our monitoring model in that it is responsible for both filtering of event reports and dissemination of the selected reports to chosen destinations [ISO 10164-5]. The components which constitute the service are shown in figure 8.2

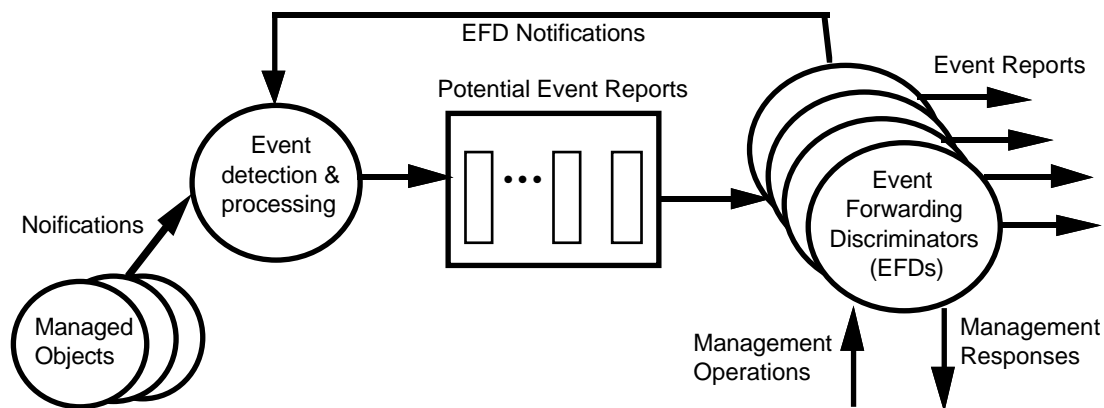


Figure 8.2 OSI Event Reporting Service

Notifications are received by a local event detection and processing component which adds the event time, originating object class and instance to the notification message to form a potential event report. These are conceptually sent to all local *Event Forwarding Discriminators (EFDs)*. An EFD holds a discriminator construct which defines the criteria used to select which of the potential event reports are forwarded to the destination address stored in the EFD. An EFD sends reports to a single destination at any time. The EFD is a managed object and so itself can generate notifications which are forwarded by another local EFD. The EFD's specify the following information:

EFD Identifier: unique identifier for the EFD

Discriminator Construct: A logical expression on attributes within the potential event reports,

e.g., $\{(dest=fred) \text{ and } (count > 50)\}$ or $\{(source=xyz) \text{ and } (size < 100)\}$

Only those potential reports which meet the criteria specified are forwarded.

Administrative, operational, and availability state information which indicates whether or not it is actually forwarding events.

Destination: primary destination to which events should be forwarded

Backup destination: a list of alternative destinations to which events reports are to be sent if the primary is unavailable.

The scheduling of event reports can be achieved by internal or external scheduling packages

Daily schedule: a list of times within a 24 hour period when forwarding is enabled

Weekly Schedule: start/ stop operating date and time, and periodic weekly schedule for enabling the event forwarding.

EFDs can be created and deleted dynamically to accomplish the registration of subscribers to a dissemination services, as described in section 4. A manager can also enable/disable the operation of an EFD and so over-ride the predefined scheduling. It is not clear whether EFDs can be used to combine lower level events to form higher level events as described in section 3.4

8.3 Log Service

Trace generation is by means of log records which can be stored in log managed objects [ISO 10164-5]. The structure of the Log service is shown in figure 8.3. The collection process receives event reports from local or remote managed objects and formats them into potential log records by adding a log record identifier and logging timestamp. A set of criteria for selecting which of these potential records are actually stored, can be specified in terms of comparison operations on data within the event reports, daily or weekly time schedules etc.

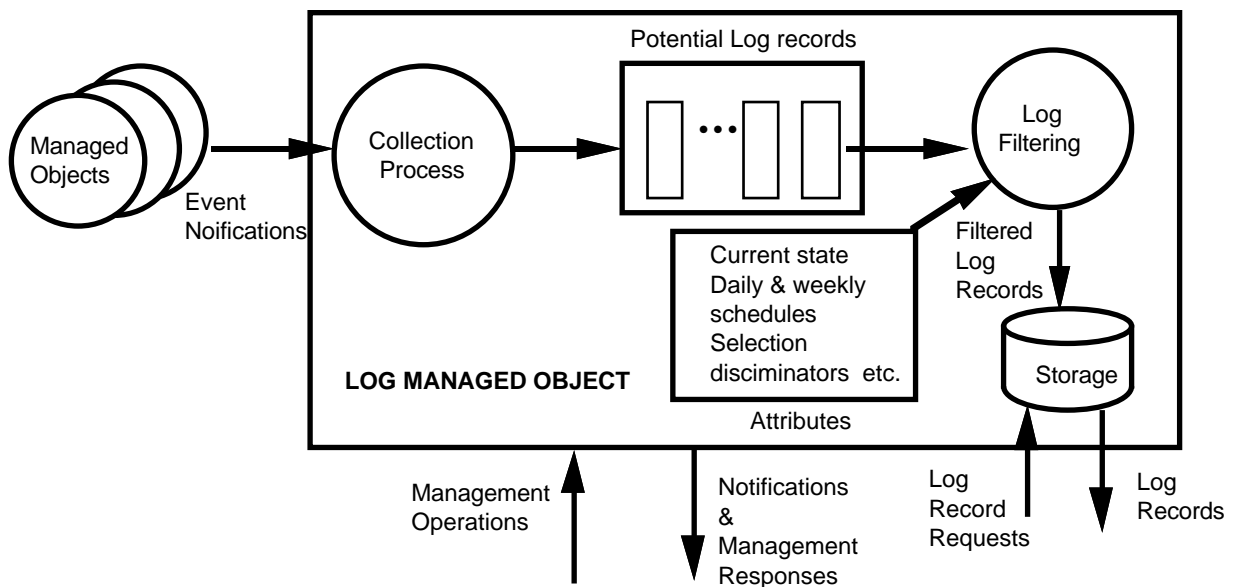


Figure 8.3 OSI Log Service

Two options are available if the log reaches maximum capacity. Either the log activity is halted, with new records being discarded or the log wraps, with the oldest records being discarded. An event report must be generated for a log halt condition and optionally to indicate a wrap condition. A capacity threshold event may be generated to warn a manager before the log is actually full. Every log must support the halt behaviour, but support of the wrap behaviour is optional.

Logs are themselves managed objects and can have their state controlled by managers for enabling/disabling the logging function. Internally defined time schedules may also change their operational state. Log attributes specify the logID, maximum number of log records, current size in bytes, number of records, log full action, the threshold for generating nearly full warning events and attributes to control the log record selection criteria. This logging service is very flexible in that managers can change the filtering criteria by changing the relevant log attributes.

8.4 Processing of Management Information

There are a number of OSI standards which specify managed objects which can be used for performing the monitoring information processing functions defined in section 3.

The OSI standard on Workload Monitoring [ISO 10164-11] defines metric objects as a means of processing monitoring information with the emphasis on statistical measurement of the performance of resources.

The simplest metric objects are counters and gauges. *Counters* increase until they reach a maximum value then wrap to zero. There may be 3 thresholds, associated with a counter. When the counter reaches the threshold value, a notification is emitted. *Gauges* can both increase and decrease in value. Two thresholds are associated with a gauge. A notification is emitted when the gauge crosses the high-level threshold (when the gauge is increasing) and similarly a notification is emitted when the gauge crosses the low level threshold (when decreasing). Only one notification will be emitted from a particular threshold until the gauge crosses the other threshold to give a hysteresis effect and stop multiple notifications being emitted if the gauge oscillates about a threshold level. As mentioned above, metric objects may periodically sample a particular attribute in a managed object in order to implement a counter or gauge with threshold notifications.

Other metric objects which have been defined include:

- Mean monitor to calculate a mean value of an attribute

- Moving Average mean monitor which uses an exponentially weighted moving average algorithm

- Mean and Variance monitor

- Mean and percentile monitor

- Mean and min-max monitor

- Scanner Summarisation Objects which provide a report on a list of attributes values of the same or different types which have been read as a single scan.

- Buffered Scanner provides a report of a list of different types of attribute values over multiple scans.

8.5 Discussion

Considerable international effort is being put into defining the comprehensive set of OSI management standards. These will support a very sophisticated monitoring service, but very few products which comply with these standards are available as yet. In addition, the number of options available and the sophistication of the service leads to some doubts as to the cost and performance capabilities of an OSI based monitoring service. The OSI standards do not define anything about the presentation of management information as this is considered an implementation issue.

9 SUMMARY

There is a need for a generic service for monitoring distributed systems as an underlying service to support all aspects of management. A monitoring service is 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.

This paper has defined a monitoring model in terms of a set of monitoring functions. This has been used 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 the monitored information to validate, combine and filter so that only relevant information is provided to clients.

The required information must be *disseminated* to those clients who have subscribed to the service and specified the particular information they require

Flexible graphical facilities are needed for the *presentation* of monitored information to human users in a form which aids comprehension and meets specific application requirements.

The emphasis of the paper has been on a survey of the current approaches to monitoring of parallel and distributed systems. However the monitoring model could be used as a framework for specifying and designing a generalised monitoring service. This is work which we intend to undertake in the future as part of a European collaborative project.

ACKNOWLEDGEMENTS

We gratefully acknowledge the support of the SERC and HP Laboratory, Bristol for a CASE studentship. 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

- [Bates 88] Bates, P., *Distributed Debugging Tools for Heterogeneous Distributed Systems*, Proc. 8th International Conference on Distributed Computing Systems, IEEE, June 1988, pp. 308-316.
- [Bemmerl et al. 90] Bemmerl, T., Lindhof, R., Treml, T., *The Distributed Monitor System of TOPSYS*, Proceedings CONPAR 1990 - VAPP IV, Sep. 1990, Springer-Verlag, pp. 756-765.
- [Bemmerl et al. 91] Bemmerl, T., Bode, A., *An Integrated Environment for Programming Distributed Memory Multiprocessors*, Proc. 2nd European Dist. Memory Comp. Conf., April 1991, pp. 130-142.
- [Chandy & Lamport 85] Chandy, K. M., Lamport, L., *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Trans. on Comp. Syst., Vol. 3, no. 7, pp. 63-75, February 1985.
- [Christian 89] Christian, F., *Probabilistic Clock Synchronisation*, Distributed Computing 3, 1989, pp. 146-158.
- [Dasgupta 86] Dasgupta, P., *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.
- [Demon 93] MARI Computer Systems Ltd, "*Demon: Distributed Environment Monitoring tool*", User's Guide and Reference Manual.
- [Duda et al 87] Duda, A., Harrus, H., Haddad, Y., Bernard, G., *Estimating Global Time in Distributed Systems*, ISEM Universite de Paris-Sud, 91405 Orsey, France, IEEE 1987, pp.299-306.
- [Feldkuhn & Erickson 89] Feldkuhn, L., Erickson, J., *Event Management as a Common Functional Area of Open Systems Management*, Proc. IFIP Sym. on Integrated Network Management, Boston 1989, pp. 365-376, North-Holland.
- [Fidge 88] Fidge, C., J., *Partial Orders for Parallel Debugging*, In Proceedings of Workshop on Parallel and Distributed Debugging, ACM, 1988, pp.183-194.
- [Haban & Wybraniec 90] Haban, D., Wybraniec, D., *A Hybrid Monitor for Behaviour and Performance Analysis of Distributed Systems*, IEEE Trans. on Software Eng., Vol. 16, No. 2, February 1990.
- [Harter et al. 85] Harter, P. K., Heimbigner, D. M., King, R., *IDD: An Interactive Distributed Debugger*, Proc. 5th Int. Conf. on Distributed Computing Systems, Denver, IEEE, pp. 498-506, May 1985.
- [Hofmann, et al. 92] Hofmann, R., Klar, R., Mohr, B., Quick, A., Siegle, M., *Distributed Performance Monitoring: Methods, Tools, and Applications*, University of Erlangen-Nurnberg, IMMD VII, Martensstrabe 3, D-8520 Erlangen, Germany.
- [Holden 88] Holden, D., et al., *An Approach to Monitoring in Distributed Systems*, Proc. Eur. Teleinformatics Conf., Vienna 1988, pp. 811-823, North-Holland 1988.
- [Holden 89a] Holden, D., Langsford, A., *MANDIS: Management of Distributed Systems*, Harwell Laboratory, U.K., Lecture Notes in Computer Science, Vol. 433, Progress in Distributed Operating Systems and Distributed Systems Management, April 1989, pp. 162-173.
- [Holden 89b] Holden, D., *Predictive Languages for Management*, Proc. IFIP Sym. on Integrated Network Management, Boston 1989, 585-596, North-Holland 1989.
- [Holden 91] Holden, D. B., *A Tutorial to Writing Programs in SESL*, Internal Report DMP/55, Sys. & S/W Eng. Grp., AEA Industrial Technology, Harwell, Jan. 1991.
- [ISO 10164-1] ISO/IEC DIS 10164-1 *Information Technology - Open Systems Interconnection - Systems Management Part 1: Object Management Function*, Oct. 1990.

- [ISO 10164-2] ISO/IEC DIS 10164-2 *Information Technology - Open Systems Interconnection - Systems Management Part 2: State Management Function*, Oct. 1990.
- [ISO 10164-4] ISO/IEC DIS 10164-4 *Information Technology - Open Systems Interconnection - Systems Management Part 4: Alarm Reporting Function*, Oct. 1990.
- [ISO 10164-5] ISO/IEC DIS 10164-5 *Information Technology - Open Systems Interconnection - Systems Management Part 5: Event Report Management Function*, Oct. 1990.
- [ISO 10164-6] ISO/IEC DIS 10164-6 *Information Technology - Open Systems Interconnection - Systems Management Part 6: Log Control Function*, Oct. 1990.
- [ISO 10164-11] ISO/IEC DIS 10164-11 *Information Technology - Open Systems Interconnection - Systems Management Part 11: Workload Monitoring Function*, April 1992.
- [ISO 10165-4] ISO/IEC DIS 10165-4 *Information Technology - Open Systems Interconnection - Structure of Management Information: Guidelines for the Definition of Managed Objects*, 1992.
- [Joyce et al. 87] Joyce, J., Lomow, G., Slind, K., Unger, B., *Monitoring Distributed Systems*, ACM Trans. Comput. Syst., Vol. 5, No. 2, May 1987, pp. 121-150.
- [Klar et al. 92] Klar, R., Quick, A., Soetz, F., *Tools for a Model-driven Instrumentation for Monitoring*, In 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.
- [Kramer et al. 89] Kramer, J., Magee, J., Ng, K., *Graphical Configuration Programming*, IEEE Computing, October 1989, pp. 53-65.
- [LaBarre 91] LaBarre, L., *Management By Exception: OSI Event Generation, Reporting, and Logging*, The MITRE Corporation, 2nd IFIP Symposium on Integrated Network Management, Washington, April 1991.
- [LeBlanc & Robbins 85] LeBlanc, R. J., Robbins, A. D., *Event-Driven Monitoring of Distributed Programs*, Proc., 5th International Conference on Distributed Computing Systems, May 1985, pp. 515-522.
- [Lamport 78] Lamport L., *Time, Clocks and the Ordering of Events in Distributed Systems*, CACM Vol 21., July. 1978, pp. 558-564.
- [Lumpp et al. 90] Lumpp, J. E., Jr., Casavant, T. L., Seigle, H. J., Marinescu, D. C., *Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems*, Proc. 10th International Conference on Distributed Systems, June 1990, pp. 476-483.
- [Magee, et al. 89] Magee, J., Kramer, J., Sloman, M., *Constructing Distributed Systems in Conic*, In IEEE Transactions on Software Engineering, Vol. 15, No. 6, June 1989, pp.663-675.
- [Malony et al. 89] Malony, A. D., Reed, D. A., Arendt, J. W., Grabas, D., Aydt, R. A., Totty, B. K., *An Integrated Performance Data Collection Analysis and Visualisation System*, Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications, 1989, pp.229-236.
- [Manning 87] Manning, C. R., *Traveler: The Apiary Observatory*, Proceedings of European Conference on Object Oriented Programming, pp. 97-105, 1987.
- [Marinescu et al. 90] Marinescu, D., C., Lump, J. E., Casavant, T. L., Siegel, H. J., *Models for Monitoring and Debugging Tools for Parallel and Distributed Software*, Journal of Parallel Distributed Computing 9, 2, June 1990, pp. 171-184.
- [Marzullo et al. 91] Marzullo, K., Cooper, R., Wood, M. D., Birman, K. P., *Tools for Distributed Application Management*, Cornell University, IEEE Computer, August 1991, pp. 42-51.

- [**McDowell & Helmbold 89**] McDowell, C. E., Helmbold, D. P., *Debugging Concurrent Programs*, ACM Computing Surveys, Vol. 21, No. 4, December 1989.
- [**Mohr 90**] Mohr, B., *Performance Evaluation of Parallel Programs in Parallel and Distributed Systems*, Proceedings CONPAR 1990 - VAPP IV, Sep. 1990, Springer-Verlag, pp. 176-187.
- [**Mohr 91**] Mohr, B., *SIMPLE: a Performance Evaluation Tool Environment for Parallel and Distributed Systems*, In A. Bode, editor, Proc. of the 2nd European Distributed Memory Computing Conference, EDMCC2, pages 80-89, Munich, Germany, April 1991. Springer, Berlin, LNCS 487.
- [**Schwarz & Mattern 92**] Schwarz, R., Mattern, F., Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail, Department of Computer Science, University of Kaiserslautern, D-6750 Kaiserslautern, Germany (1992).
- [**Shim & Ramamoorthy 90**] Shim, Y. C., Ramamoorthy, C. V., *Monitoring and Control of Distributed Systems*, Proceeding of First International Conference on Systems Integration, Morristown, NJ, IEEE Computing Press, April 23-26, 1990, pp. 672-681.
- [**Sloman 87**] Sloman, M., *Distributed Systems Management*, Imperial College Research Report, DOC 87/6, 1 April 1987.
- [**Snodgrass 88**] Snodgrass, R., *A relational Approach to Monitoring Complex Systems*, ACM Trans. on Computer Systems, vol., 6, no. 2, pp. 157-196, May 1988.
- [**Socha et al. 89**] Socha, D., Bailey, M., L., Notkin, D., *Voyeur: Graphical Views of Parallel Programs*, In Proceedings of Workshop on Parallel and Distributed Debugging, May 5-6, 1988, SIGPLAN NOTICES, Vol. 24, Number 1, Jan. 89, pp. 206-215.
- [**Spezialetti & Kearns 89**] Spezialetti, M., Kearns, J. P., *Simultaneous Regions: A Framework for Consistent Monitoring of Distributed Systems*, Proc. 9th Intl. Conference on Distributed Computing Systems, pp. 61-68, 1989.
- [**Stone 88**] Stone, J. M., *A Graphical Representation of Concurrent Processes*, Proceedings of Workshop on Parallel and Distributed Debugging, ACM Published as SIGPLAN Notices 24, 1, January 1989, pp. 226-235.
- [**Tsai et al. 90**] Tsai, J. J.-P., Fang, K.-Y., Chen, H.-Y., *A Non-invasive Architecture to Monitor Real-Time Distributed Systems*, IEEE Computer 23, 3, March 1990, pp. 11-23.
- [**Van Riek & Tourancheau 91**] Van Riek, M., Tourancheau, B., *A General Approach to the Monitoring of Distributed Memory Machines - A Survey*, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Supérieure de Lyon, Institut des Sciences de la Matière de l'Université Claude Bernard de Lyon, Institut IMAG, Unité de Recherche Associée au CNRS No. 1398, Research Report No. 91/28, September 1991.
- [**Wybraniec & Haban 90**] Wybraniec, D., Haban, D., *Monitoring and Measuring Distributed Systems*, Performance Instrumentation and Visualisation, ACM Press, ISBN 0-201-50937-7 - 1990, pp. 27-45.
- [**Wolfson et al. 91**] Wolfson, O., Sengupta, S., Yemini, Y., *Managing Communication Networks by Monitoring Databases*, IEEE Trans. on S/W Eng., Vol. 17, No. 9, Sept. 1991.