Monitoring of Distributed Systems

Y.C. Shim
C.V. Ramamoorthy

Computer Science Division
Dept. of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

Abstract

Distributed systems offer opportunities for attaining high performance, fault-tolerance, information sharing, resource sharing, etc. But we cannot benefit from these potential advantages without suitable management functions such as performance management, fault management, security management, etc. Underlying all these management functions is the monitoring of the distributed system. Monitoring consists of collecting information from the system and detecting particular events and states using the collected information. These events and states can be symptoms for performance degradations, erroneous functions, suspicious activities, etc. and are subject to further analysis. Detecting events and states requires a specification language and an efficient detection algorithm. In this paper we introduce an event/state specification language based on classical temporal logic and a detection algorithm which is a modification of the RETE algorithm for OPS5 rule-based language. We also compare our language with other specification languages.

1. INTRODUCTION

With increasing availability of distributed computing environments and communication networks, people have been paying much attention to the issue of the effective utilization of these systems with the goal to make the most of the potential benefits that these systems can offer such as high performance, high fault-tolerance, information sharing, resource sharing, etc.

In a distributed computing environment one can develop a program consisting of many processes which are running on different processors. To debug the program and improve the performance of the program, he should run the program, monitor its behavior to find errors and performance bottlenecks, and correct these problems. And in a distributed computing environment processes of many different programs and even processes of a single program can compete for limited computing resources such as processors. To improve the throughput of the whole system and also the performance of a single program, it is important to monitor the utilization of each processors and redistribute the load of processors if required.

A communication network forms the backbone of distributed systems and makes it possible to exchange information and access remote resources. In a communication network, the load, which is the amount of requests for the message delivery, is varying, the topology of the network changes because the nodes and links can fail and recover, and noises and malicious intrusions can prevent the reliable and secure delivery of messages. To be able to deliver the messages with tolerable response time and without errors, it is very important to monitor the network to find any performance degradation, failures, errors, and suspicious activities and then make changes in the routing, the network configuration, the parameters in the protocol, etc.

From these observations it is clear that monitoring is a very essential function for the effective utilization of the distributed systems. Basically monitoring consists of two activities: the collection of data from the monitored system and the analysis of the collected data. Data such as events and states are collected by the probes installed in the system. Informally an event is defined to be something that occurs in the system such as a process creation and a state is defined to be a property that holds in the system such as a load average. The amount of data generated by the probes is very large and one is usually interested in particular events, states, or logical combinations of them. Here we use the terminology an event/state to designate an event, a state, or any combination of events and states. In the analysis those interesting events/states are specified using a certain specification language and detected by using the collected data. The specification language should be expressive enough so that one can represent many different types of events/states easily. Typically an event/state consists of many component events and states. It should be possible
to have any logical combination of those component events and states and represent arbitrary temporal relations among them. Also the specification language should be designed in such a way that any events/states specified by the language should be detected efficiently from the large amount of data received from the probes. In this paper we introduce an event/state specification language based on classical temporal logic and an efficient detection algorithm for the events/states specified by our specification language. The proposed algorithm is a variation of the RETE algorithm for the rule-based language, OPS5.

The rest of the paper is organized as follows. In the section 2 we describe the overall structure of our monitoring system. In the section 3 our event/state specification language is introduced. The detection algorithm is presented in the section 4. The section 5 compares our specification language with other languages and is followed by the conclusion.

2. DESCRIPTION OF THE OVERALL SYSTEM

In this section we discuss the components of the monitoring system. Figure 1 is the diagram of the system indicating the major components and the interconnections among them. The figure illustrates just the logical structure of the system and the physical system can be distributed over the distributed environment.

![Figure 1. Overall structure of the monitoring system](image)

Data are collected by probes installed in the monitored system. There are two kinds of probes: tracing probes and sampling probes. A tracing probe is invoked when an operation such as a process creation, a file access, a message sending, etc. occurs and collects information about that operation. Each information collected by a tracing probe is encapsulated in a data packet including the operation name, a list of attribute values for the operation, a start time, and an end time. It takes the general form as follow:

\[(\text{operation-name attr}_1 \text{ attr}_2 \cdots \text{ start-time end-time})\]

For example, if a process with an id 1123 on the host mars creates a child process with an id 834 on the host venus and the process creation takes from the start time 11457 to the end time 11469, then the generated data packet will look like:

\[(\text{create-process mars 1123 venus 834 11457 11469})\]

A sampling probe is periodically or aperiodically invoked and reads the value for a time-varying state of an object or a relationship among objects in the monitored system such as a load average of a host. The data packet made by a sampling probe includes the name of the object or relationship, the unique identifier for the certain instance of the object or relationship, the sampled state name and its value, and the sampling time. It has the general form as follow:

\[(\text{object-relationship-name, unique-id},\]
\[(\text{state-name, sampled-value, sampling-time})\]

If a sampling probe samples the value for the load average of the host venus at the time 19234 and the value is 1.3, then the generated data packet becomes:

\[(\text{host venus (load-average 1.3) 19234})\]

The temporal information in the data packets can be obtained from either a logical clock or a physical clock. A logical clock defines the partial ordering of events occurring in a set of processes communicating with each other[1]. The logical clock is essential for understanding the behavior of distributed cooperating processes and debugging them but cannot provide information such as the wall clock times at which an event started and ended and the length of the time interval during which a state has been holding. These types of temporal information are crucial to monitoring and controlling distributed systems and therefore we adopt a physical clock. We assume that well-synchronized physical clock is provided by a global clock implemented by a distributed algorithm and available to each processor.

The user interface receives requests from the users, forwards them to the appropriate component, and presents the results to the users. One can request the detection of
an event/state by submitting the description of that event/state written in an event/state specification language. One can also create or destroy a relation in the database and read or modify the database content by giving a command written in a database manipulation language.

The event/state detector receives an event/state specification from the user interface, compiles it to check the syntactic and semantic correctness, and stores it until it is detected. The detector also receives the data collected by the probes, check them against the stored specifications, sends a reply to a user through the user interface when an event/state is detected. A detection of an event/state implies the occurrence of something interesting, undesirable, or suspicious in the monitored system. Upon receiving the notification of the detection, one can take an appropriate control action or make a knowledge-based analysis of the detected event/state.

The data collected by probes are stored in the database. The database can answer queries on the system status but also be the basis for the knowledge-based analysis of events/states and the knowledge-based planning of control actions. The collected data have timestamps and one is usually interested in the temporal relationships among many events/states. In our monitoring system we adopt a history database where temporal information of events/states can be stored and the queries including temporal relationships can be answered efficiently[2]. There are two types of basic relations: an operation relation and a state relation. An operation relation keeps information on the occurrences of one type of operation supplied by tracing probes. Figure 2.a is an example of an operation relation for the process creation operation. A state relation contains constant or time-varying information on the states of an object or a relationship among objects. The information for a state relation is provided at the initial configuration time of the monitored system or during the operation time by sampling and tracing probes. Figure 2.b is an example of a state relation for the host object. A relation can be created and destroyed by giving a proper data definition command. A command to create relation should include the name of the relation, the name and type of attributes, and the type of the relation, either operation or state. If it is an operation relation, then corresponding tracing probes are enabled. It it is a state relation, then one should provide further information on how each attribute value can be obtained, by either tracing probes or sampling probes. Then the corresponding tracing and/or sampling probes are enabled. When a relation is destroyed the corresponding probes are disabled. Also using a data manipulation language one can read and modify the content of the history database. One can think of both an event/state detection request and a read request to the history database as the same type of query to the data collected from the monitored system. But the basic difference is that an event/state detection request is a query to the data collected after the submission of the query and a read request to the history database is a query to the data collected before the submission of the query. Due to this difference the algorithms to process these two types of queries also become different.

In this paper we focus on the specification language of events/states and its detection algorithm.

3. SPECIFICATION LANGUAGE

First we list what events/states we are interested in. We are interested in events such as an occurrence of an operation and state transition (entering a new state or leaving an old state) and states such as current state, past state, statistics of state over a time interval. Also we are interested in the logical combination of events and states along with comparison over time domain so that we specify their temporal relationships.

Temporal logic is a formalism for reasoning about a changing world. Depending upon the way in which temporal information is incorporated temporal logic can take one of two forms: classical temporal logic and model temporal logic. In classical temporal logic the temporal information is explicitly represented by specifying time points or a time interval to each entity that we want to describe while the temporal relationship among entities is represented by using modal operators such as always, eventually, next, etc. in modal temporal logic. We adopt

<table>
<thead>
<tr>
<th>NAME</th>
<th>MACHINE-TYPE</th>
<th>LOAD-AVG</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emie</td>
<td>Vax</td>
<td>1 → 2.0</td>
<td>1 → up</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 → 3.5</td>
<td>3 → down</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 → null</td>
<td>5 → up</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 → 1.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 → 1.0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. History database
I us OR I evenustate

FORALL I paranormal qualified states and events are combined to make a compound event/state and temporal relationships among them can be specified using operators such as AND, OR, MINUS, FORALL, and EXISTS.

In the rest of this section we explain the syntax and semantics for the operands and operators with examples. State type operands represent current states, past states, or aggregated states. Some examples are

1. (host (hostname ernie) (la 10))
2. (part-of (hostname mars) (clustermame xcssun))
3. (host (hostname ernie) ((la 5) 10))
4. (host (hostname ?x) ((la max 5) 10))

The example 1 is a current state of an object such that a host object with a name ernie has load average greater than 10. The example 2 is a current state of a relationship such that the host mars has a part-of relationship with the cluster xcssun. The example 3 is a past state such that a host object with a name ernie has load average greater than 10 five time units ago and the example 4 is an aggregated state such that a host object with some name ?x has maximum load average greater than 10 during the last five time units. A symbol starting with '?' such as ?x is a variable. Following is an example of an operation type operand.

(login (hostname arpa) (acctname john))

This specifies an occurrence of an operation login such that account John logs in the host Arpa.

Using operators constructing a compound state one specifies the logical relation among component states. These component states can have variables and the condition that should be satisfied among these variables can be specified. Because one cannot specify any temporal information about a compound state all the variables appearing in a compound state are nontemporal variables. The syntax for a compound state (cstate) along with their meaning is as follow:

cstate := (SAND cstate1 cstate2 cond) /* both cstate1 and cstate2 are satisfied */ /* and variables in cstate1 and cstate2 */ /* satisfy cond */
1 (SOR cstate1 cstate2 cond) /* either cstate1 or cstate2 is satisfied. */ /* And cstate1 and cstate2 have the same */ /* set of variables which satisfy cond */
Two expressions \( st \) and \( et \) are temporal expressions and represent time points. Variables can appear in the temporal expressions and they are called \textit{temporal variables}. But the temporal variables in a temporally qualified state have different semantics from those in a temporally qualified event. We explain the difference with examples.

1. \[ \text{(HOLD } {?t1} \mathbin{?t2} \text{ (host (hostname ernie)) (la > 10)}) \]
2. \[ \text{(OCCUR } {?t3} \mathbin{?t4} \text{ (login (hostname arpa)) (acctname johni)}) \]

If we have a report from a sampling probe that the load average of the host \textit{ernie} has been higher than 10 between 1130 to 1530, then \(?t1\) and \(?t2\) can take any value between 1130 and 1530 only with the constraint that \(?t1 \leq ?t2\). This is because the temporally qualified state in the example 1 is true during any subinterval of the time interval bounded by 1130 and 1530. But if we have a report from a tracing probe that John logged in the host arpa and the login process took from 1234 to 1238, then \(?t3\) gets 1234 and \(?t4\) 1237 because the temporally qualified event in the example 2 is true only if \(?t3 = 1234\) and \(?t4 = 1237\). The temporal variables in the temporally qualified state are called \textit{range type} temporal variables and those in the temporally qualified events \textit{point type} temporal variables.

The syntax for an event/state (es) is given below. The explanation of the meaning for the operators is omitted because their meaning can be easily understood.

\[ \text{es ::= (AND es es cond)} \]
\[ \text{(OR es es cond)} \]
\[ \text{(MINUS es es cond)} \]
\[ \text{(FORALL (vars) es es cond)} \]
\[ \text{(EXISTS (vars) es es cond)} \]
\[ \text{temporally-qualified-state } ::= \]
\[ \text{(HOLD st et compound-state);} \]
\[ \text{temporally-qualified-event } ::= \]
\[ \text{(OCCUR st et operation-type-operand)} \]
\[ \text{(OCCUR st et compound-state)} \]
\[ \text{state-type-operand} ; \]

The expression \textit{cond} is a constraint that should be satisfied by variables appearing in the component events/states. The difference between \textit{cond} in an event/state and that in a compound state is that the former can have both nontemporal and temporal variables whereas the latter can have only nontemporal variables. There are some restrictions in the way that temporal variables can be used in an event/state. A range type temporal variable cannot appear as a free variable in the component events/states of operators \textit{OR, MINUS, FORALL, and EXISTS}. A variable is defined to be a free variable in a logical expression if the variable is neither universally nor existentially quantified in the logical expression. And if a temporal variable appears in both component events/states of \textit{AND} operator, then in at least one component event/state it should be a point type. The purpose of these restrictions on the usage of temporal variables and the reason for having operators for compound states

252
separate from operators for events/states are to simplify
the semantics of operators and therefore make it possible
to have an efficient detection algorithm. But in this paper
we avoid detailed explanation. Some examples for
events/states are

1. \[
\text{(HOLD } ?t-5 \text{ ?t)} \\
\text{(SFORALL } (?x) \\
\text{(SAND} \\
\text{(part-of (hostname } ?x)) \\
\text{(clustername xcssun))} \\
\text{(host (hostname } ?x) (status up))) \\
\text{(host (hostname } ?x) (la > 10))))
\]

2. \[
\text{(MINUS} \\
\text{(AND (OCCUR } ?t1 \text{ ?t1} \\
\text{(send (sender A) (receiver B)))} \\
\text{(OCCUR } ?t2 \text{ ?t2} \\
\text{(send (sender C) (receiver D)))} \\
\text{(?t1 ≤ } ?t2 \text{ ≤ } ?t1+10)) \\
\text{(OCCUR } ?t3 \text{ ?t3} \\
\text{(send (sender B) (receiver C)))} \\
\text{(?t1 ≤ } ?t3 \text{ ≤ } ?t2))
\]

The example 1 is an event/state such that for all the hosts
which belong to the cluster xcssun and are up, the load
average is higher than 10 for consecutive 5 time units.
This can be a symptom of a cluster overload which may
require the migration of some of the processes running on
xcssun to some other clusters. The example 2 is an
event/state such that the process A sends a message to the
process B then the process C sends a message to the pro-
cess D within next 10 time units but the process B does
not send a message to the process C in between. This can
indicate an incorrect communication pattern among
processes and require an analysis of the program under
monitoring.

4. DETECTION ALGORITHM

In this section we describe the algorithm for detect-
ing events/states specified with our language. The algo-
rithm is based upon the RETE algorithm for OPS5 rule-
based language[5,6]. We first describe the basic ideas for
the detection algorithm. We view an event/state
specification as a tree where a leaf node is an operand and
an inner node is an operator as in the figure 4. For each
type of probes for operations, object states, and relation-
ship states, a list of pointers to operands is maintained.
There is one list for probes sampling the load average,
one list for probes tracing login operations, etc. And in a
list there exists a pointer to a certain operand, if data
received from the probes represented by the list can affect

The overall structure for event/state detection

```
AND

HOLD ?t1 ?t2

SAND

(host (hostname ernie) (la > 5))

pointer list for probes sampling load average

OCCUR ?t2 ?t2

AND

OCCUR ?t3 ?t3

(login ...)

pointer list for probes tracing login operation
```

Figure 4. The overall structure for event/state detection
the truth value of the operand. Whenever a new data packet arrives from a probe, the operands, which are pointed by the pointers in the list corresponding to the type of the probe, are checked against the content of the data packet. If the truth value of an operand changes, then this change is moved up the specification tree as high as possible. Partial detection results are stored in some of the inner nodes so that any previous data need not be checked. And the pointer list mechanism makes it possible to process only those data reports that may affect any events/states and also process only those events/states that may be affected by those data packets.

In the development of the detection algorithm we make following assumptions. We assume that there exists a well-synchronized global clock so that we can determine the temporal ordering and duration based upon the timing information included in every data packet from probes. There is non-uniform but small and finite communication delay, $T_{\text{max}}$. And data packets from the probes arrive at the detector in the order of their collection time. This assumption can be made possible by forwarding a data packet with a collection time $t$ to the detector at $t+T_{\text{max}}$.

But there is a problem when we implement this type of algorithm. Let's take an inner node SAND in the figure 5 as an example. It has two subtrees. The left subtree receives data including $?y$ value, processes them according to the specification corresponding to the left subtree, and sends up $?y$ value to the SAND node. Likewise the right subtree sends up $?x$ value. Because there is a finite processing time, data from two subtrees cannot arrive at the SAND node simultaneously. Let's assume that data from the right subtree always arrives earlier than that from the left subtree. At $t = T1$ the right subtree sends new $?x$ value to the SAND. As soon as the SAND node receives the data from the right subtree, it is processed and generates the correct output. At $t = T2$ not only the right subtree but also the left subtree sends new data. Due to our assumption $?x$ value from the right subtree arrives earlier and is processed in the SAND node to generate an incorrect output, $?x=3$ and $?y=2$, as in the figure. Later when the $?y$ value from the left subtree is received and processed in the SAND node, this incorrect output will disappear and the correct output will be generated. The same kind of problem exists even if the output from the left subtree is processed first. From this example we conclude that there should be a mechanism that informs a binary operator of whether it will receive a data with the same timestamp from one of its subtrees when it has received data from the other subtree. If a binary node receives a data from one of its subtrees but the other subtree does not have any data with the same timestamp, it can be processed. But if the binary node receives a data from one of its subtrees and the other subtree has data which have the same timestamp but have not been sent to the binary node yet, then it should wait for the data from the other node before being processed.

We solve this problem with a branch marking method. A branch between two nodes is marked as one of three marks: NC (No Change), MC (May Change), and R (Ready). If a branch is marked as NC, then there is no data change from the subtree connected by this branch. If a branch is marked as MC, then there may be a change of data from the subtree connected by this branch. If a branch is marked as R, then both the data change from the child subtree connected by this branch arrive at the node at the same time. And branches are marked by following rules.

1. Initially all the branches are marked as NC.
2. If all the child branches of a branch are marked as NC, then it is marked as NC.
3. If a branch was marked as MC and its child node is processed but does not generate any data, then it is marked as NC.
4. If a branch has a leaf node as a child node and there

---

Figure 5. Problems with event/state detection
is a data packet from a probe that may change the truth value of the leaf node but the leaf node is not processed yet, then the branch is marked as $MC$.

5. If a branch has a child branch marked as $MC$, it is marked as $MC$.

6. If a branch was marked as $MC$ and its child node is processed and generates new data, then it is marked as $R$.

And an inner node can be processed when following two conditions are satisfied.

1. At least one child branch is marked as $R$.
2. No child branch is marked as $MC$.

These two conditions guarantee that only those inner nodes all of whose subtrees have been processed and sent up the data are processed.

With the above rules the detection algorithm at time $t = T + T_{\text{max}}$ is as follow.

1. Forward all state/operation reports with collection time $T$ to the detector while putting all the leaf nodes that may be affected by these data reports in the queue
2. Mark all the ancestor branches of leaf nodes in the queue as $MC$
3. while (queue is not empty) {
   3.1. Take a node from the queue and process
   3.2. Change branch marking and put new processable nodes in the queue if any
}

5. COMPARISON WITH RELATED WORKS

For the specification of events/states either some general purpose languages have been used or special purpose languages have been developed some of them are as follow:

1. OPS5: OPS[6] is a rule-based language and its left hand side is used to specify event/states.
2. History Database Query Language: Snodgrass[7] developed a query language for his history database and used it to specify events/states in distributed computation.
3. Extended regular expression: Many researchers[8,9,10] developed specification languages based upon extended regular expression which is a regular expression extended with a shuffle operator and used them for distributed debugging.
4. Network Event Manager (NEM): In the NEM which is a monitoring and control tool for distributed systems[11] an event/state specification language based upon propositional logic was developed.

First we compare our language with the above languages informally. Following are the list of events or states that we would like to specify with a language.

R1: Current state and past state
R2: Aggregated state (statistics over time domain or object space)
R3: Same state over a group of qualified objects (e.g., forall hosts which belong to the cluster xcsun, the load average is greater than 10)
R4: Occurrence of operation
R5: State transition
R6: Missing event
R7: Combination of event and state
R8: Temporal relation among events and states

Each language can specify events and states listed above as follow:

- OPS5 $\rightarrow$ R1, R4-R8
- Snodgrass' history database query language $\rightarrow$ R1, R4, R5, R7, R8
- Extended regular expression-based language[8] $\rightarrow$ R4, R8(limited)
- Extended regular expression-based language[9] $\rightarrow$ R4, R7(very limited), R8(limited)
- Extended regular expression-based language[10] $\rightarrow$ R4, R5(limited), R7(limited), R8(limited)
- NEM $\rightarrow$ R1, R2, R5
- Our language $\rightarrow$ R1-R8

OPS5 can specify most of the events and states but it is impossible or very cumbersome to specify compound states. Snodgrass' history database query language used for monitoring distributed systems has only limited set of operators from the relational algebra and its representation power is limited. Moreover in his language all the events are considered to occur instantaneously, so an event whose occurrence takes non-zero time interval cannot be specified. All the languages based on the extended regular expression put emphasis on the temporal ordering
of events and have no or very limited capability to specify states. In these languages events are assumed to occur instantaneously. The language of NEM is mainly for specifying compound states and state transitions and the representation of occurrence of operations and temporal relationships among states and events are not considered.

6. CONCLUSION

Monitoring is an essential function for the management of distributed systems. It consists of two subfunctions: collecting information from the system and detecting particular events and states from the collected information. Probes should be installed and enabled in the system to collect information. To detect events and states a specification language to represent them and an efficient algorithm are required.

In this paper we introduced a specification language based on classical temporal logic. This language can be easily used to specify states, operations, state transitions, logical combinations of them along with their temporal relationships. It is also amenable to an efficient implementation and a detection algorithm was presented. The algorithm is based upon the RETE algorithm for the OPS5 rule-based language but deals with more operators, consumes less memory space for the detection of certain events and states, and solves some timing problems.

The described monitoring system is implemented on a network of 8 Sun microsystems workstations connected by an Ethernet at University of California, Berkeley. It consists of 8k lines of C code and the detection algorithm is shown to be efficient with experimentation.

References

7. R. Snodgrass, "A Relational Approach to Monitor-