*Daren A. Reed and Kenneth J. Turner. Components for Quality of Service in Distributed Environments: Monitoring Service (extended version). In Andrew Campbell and Klara Nahrstedt, editors, Proc. 5th International Workshop on Quality of Service, pages 255-258, Columbia University, New York, USA, May 1997.*

# Support Components for Quality of Service in Distributed Environments: Monitoring Service

D.A. Reed (dar@cs.stir.ac.uk) and K.J. Turner (kjt@cs.stir.ac.uk)
Department of Computing Science and Mathematics, University of Stirling
Stirling, FK9 4LA, UK

A.P. Grace (apgrace@jungle.bt.co.uk)
Distributed Systems Group, BT Laboratories
Martlesham Heath, Ipswich, IP5 7RE, UK

*Abstract* - A number of services are required in a Quality of Service oriented distributed environment. One of these required services is monitoring. To provide a complete picture monitoring must occur with both local and global scope. Rather than have a bespoke monitoring system for each distributed application it is proposed that the optimum solution is a generic service which can be modified to suit the distributed application. The approach presented in this paper develops naturally from formal system specifications and encompasses many of the Quality of Service characteristics outlined in the proposed ISO Quality of Service Framework Standard.

*Keywords* - Quality of Service, monitoring, distributed environments

## 1. Introduction

There are many 'middleware' distributed systems currently in use around the world including: the Distributed Computing Environment[1] (DCE), ANSAware[2] and a number of Common Object Request Broker Architecture[3] (CORBA) implementations. Until recently Quality of Service (QoS) was not a vital issue in the design of these systems but since the introduction of multimedia services with time critical characteristics and the need for synchronisation between data streams this has changed. Most of these 'middleware' systems have been expanded to encompass QoS; a notable exception being DCE although the Open Software Foundation (OSF) is investigating other QoS initiatives for 'middleware'. Architecture Projects Management (APM) for example is defining the Distributed Interactive MultiMedia Architecture (DIMMA) **[1-3]** as an extension of the real-time Advanced Network System Architecture (ANSA) model [4-6]. Methodologies which introduce QoS management into existing systems are currently of interest to 'middleware' providers.

This paper outlines a generic QoS monitoring service which is tailored by the distributed application. The methodology progresses naturally from formal temporal specification techniques thereby aiding development and design of the distributed application. Also described are a number of further support components that a designer/programmer might expect in a QoS-oriented distributed environment.

Section 2 briefly describes other services necessary for a QoS-oriented distributed environment. It includes the responsibilities of these other services with respect to the monitoring service and to the client distributed applications. The descriptions are by necessity brief. Section 3 outlines in detail the proposed monitoring service. Section 3.1 explains the motivation for monitoring and the functional and performance requirements demanded by a

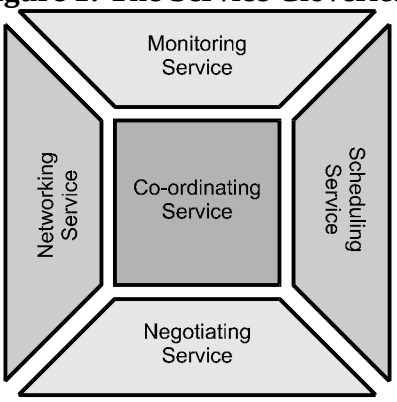
[1] URL, http://www.osf.org
[2] URL, http://www.ansa.co.uk
[3] URL, http://www.omg.org

client distributed application of a monitoring service. Section 3.2 describes the structure of the monitoring service. Section 3.3 explains how QoS characteristics can be expressed while section 3.4 extends this relationship to include characteristics outlined in the ISO QoS framework document [7]. Section 4 takes a simple example and provides a QoS monitoring schema for it. The paper concludes with an assessment of this methodology including view on further work and development.

## 2. Service Components

The services proposed for supporting QoS within QoS-oriented distributed environments are co-ordinating, monitoring, negotiating, networking and scheduling. Figure 1 shows the relationship between the five services. The co-ordinating service provides the pivotal role, interfacing with the other four services. Monitoring and negotiating services need not interface directly; negotiation deals with establishment concerns, monitoring deals with operational concerns. Networking and scheduling services are similarly non-contiguous; scheduling deals with local resources, networking deals with remote communications. The negotiating service also establishes the distributed application's QoS requirements of the infrastructure so it must interface with the networking and scheduling services during the establishment phase of the distributed application, just as the monitoring service must interface with these services during the operational phase of the distributed application.

**Figure 1: The Service Cloverleaf**




A descriptive outline of the four other services is required since these services do not operate in isolation. The descriptions refer to clients. A client is taken mean a component or group of components which require QoS support regardless of whether the component performs a client or server role within the distributed application.

The co-ordinating service is a facilitator. All client distributed applications initially deal with the co-ordinating service. This service is responsible for organising the other services for the client distributed application.

The negotiating service is an arbitrator. It establishes QoS characteristics between the components within the client distributed application. The negotiating service also negotiates the QoS characteristics required of the infrastructure, negotiating with both the networking and scheduling services.

The scheduling service is a resource manager. It establishes and manages local resources which possess the QoS characteristics required by the client distributed applications. These resources include available local memory, peripherals and CPU cycles. The scheduling service operates as an access control mechanism. In future work, the scheduling service will communicate with 'intelligent adaptive' QoS-oriented resource components thereby reducing the prediction required to maintain a reliable service to the client distributed applications.

The networking service is a transport manager. It establishes and manages communication resources which possess the QoS characteristics required by the client distributed applications. The networking service works closely with the negotiating service to provide the required communications infrastructure. The networking service is largely predictive operating as an access control mechanism. In future work, the networking service will communicate with 'intelligent adaptive' QoS-oriented network components thereby reducing the prediction required to maintain a reliable service to the client distributed applications.

# 3. Monitoring Service

## *3.1 Motivation*

Monitoring is an essential service in a QoS-oriented distributed environment. After established ranges for QoS characteristics during the establishment phase (negotiating service) it is necessary to monitor these QoS characteristics during the operational phase (monitoring service). This monitoring is necessary because distributed systems are by nature mathematically chaotic. In a completely managed system, operating full guarantees with respect to resource availability, monitoring might not be necessary. Such utopian systems can exist but require that resources stand idle just in case they are required by a current component. A monitoring service is required to provide information to counter the effects of chaos in the system.

A large number and varied range of distributed applications have QoS characteristics which require monitoring. If each distributed application had a unique monitoring service this would increase the size and complexity of the distributed application and could lead to resource deprivation, possibly precipitating a failure which would not have occurred if management overheads been lower. A generic service has lower management overheads since only one copy is required by all distributed applications within the distributed environment.

Monitoring can also have additional benefits, particularly in the predictive QoS of a distributed application. Statistics created from previous run-time occurrences of a particular distributed application can highlight weaknesses in the QoS requested with respect to the infrastructure. Alternatively performance comparisons between different software components with identical functions becomes possible given sufficient statistics which may also influence the distributed application.

There are a number of requirements of a monitoring service: minimum interference with the application, minimum operational resource overhead, timely response to erroneous conditions, correct response to erroneous conditions. These requirements are easily explained although some will be shown to be antagonistic.

The monitoring service will inevitably perturb distributed applications from normal operation. A distributed application that spends valuable time informing the monitoring service about its operation may lead to a 'bureaucracy breakdown'; the distributed application fails to meet its requirements because it spends time interfacing with the monitoring service. This must be minimised thereby reducing the effect.

The monitoring service will also consume system resources. In order that distributed applications are not unnecessarily perturbed the system resource consumption should be minimised and regular. Regrettably system resource consumption is related to the timeliness of response to erroneous conditions in the distributed application. Consumption increases as faster responses are required.

A monitoring service which correctly identifies all erroneous conditions and the cause but does not report before the distributed application fails is worthless. The monitoring

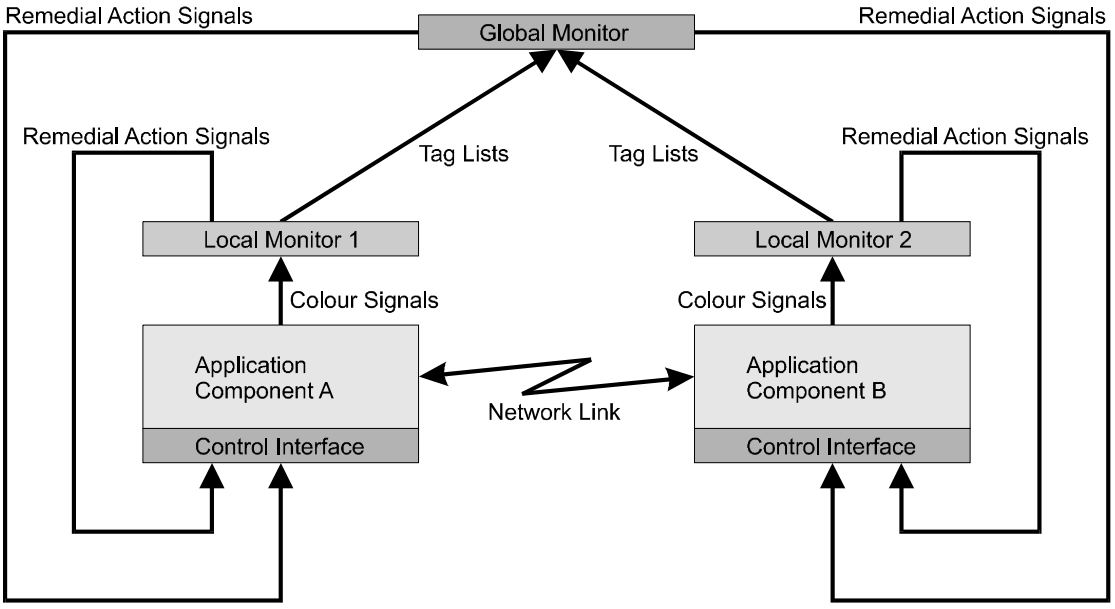
service must provide timely reports regardless of whether the distributed application can perform remedial action to rectify the failure. Timeliness is subjective and varies from application to application. Being timely also impacts on system resources. A faster response time will require the monitoring service to distribute monitored information more often thereby increasing the network overhead.

Correct response to erroneous conditions is not truly a monitoring service requirement. The monitoring service simply evaluates the status of conditions of importance to the distributed application. The distributed application designer must compose a monitoring schema which will reflect the required QoS characteristics.

### *3.2 Elements*

The monitoring service proposed in this paper is based on a 'witness' monitor. It is not built into the distributed application and is designed to reduce the interference with the client distributed application. The monitoring service receives *signals* from the distributed application and combines these *signals* with a number of pre-stated conditions established during negotiation which determines if a QoS transgression has arisen. The monitor then informs the client distributed application of the transgression. How the distributed application proceeds is not the responsibility of the monitoring service. Figure 2 gives a graphical representation of this arrangement.

**Figure 2**




In order for the monitoring service to be generic there is no application specific information conveyed to it in the *signals* from the client distributed application. The conditions presented to the monitor encapsulate the required specific QoS characteristics in a general notation. The monitoring service can also monitor the infrastructure supporting the distributed application provided it has been designed to integrate with the monitoring service. The infrastructure is treated as simply another part of the distributed application.

This monitoring service has been being developed with application pair-wise events as the underlying model. In a pair-wise event there is an action event and a reaction event. The reaction event occurs after the action event and is causally linked. For the purposes of this paper the ratio of action to reaction will be taken as 1:1. This is not a limitation of the model since the 1:n case is a simple extension of 1:1. The n:1 case, where a number of related action

events precipitate a single reaction event, and the n:m case, where a number of related action events precipitate a number of related reaction events, are considered unlikely to be required.

The client distributed application sends *signals* to the monitoring service; these *signals* are indicative of events within the distributed application. The *signal* is characterised by a *colour*. This terminology is used to avoid the overloading of the term 'channel'. Each *colour* is associated with an single type of event or related set of events within a component of the client distributed application. The *colour* is a unique identifier which can be used by the client distributed application to state the QoS conditions the monitoring service is to observe.

Each 'node' with component of the distributed application requiring QoS monitoring runs a local monitoring service component. Co-location is required to minimise latency between the event occurring within the distributed application and the event being registered as occurring at the monitoring component. Local monitors receive *signals* from the client applications which they combine with a *time-stamp* to create an *event tag*. This *event tag* is then used to test *local conditions*. *Event tags* are associated together according to *colour* to create an *event tag list*. When necessarily, this is periodically sent to the global monitor.

A *signal* is composed of a *colour* identifier, *sequence* number and *signature*: *signal* ≡ (*colour, sequence, signature*). Signals are only sent between the client distributed application and the local monitor. The *sequence* number is required to re-sequence event pairs at the reaction event. The *signature* is an application derived 'label' which identifies the data at the monitored event. It needs to be reasonably unique with respect to the data and is required for simple error checking. Although the underlying transfer protocol may support error checking/recovery and re-sequencing, the generic nature of the monitoring service means that this cannot be assumed. In the event of an error occurring, the monitoring service will not attempt recovery but will use this information in evaluating the QoS conditions. The underlying protocol may of course act upon its own internal error check and re-sequencing procedures.

An *event tag* is the combination of a *signal* and a *time-stamp* indicating when the *signal* arrived at the local monitor: *tag* ≡ (*signal, time-stamp*). The *time-stamp* is attached by the monitor for a number of reasons. Firstly the monitor is trusted to be honest: it will attach the *time-stamp* on receipt of the *signal*. This precludes the possibility of components 'fixing' the conditions. Secondly the monitoring service requires a global clock, although this clock need not be perfectly synchronised between all nodes of the client distributed application. The service must be capable of establishing the difference between these clocks to at least one order of magnitude less than the accuracy required by the client distributed application.

An *event tag list* is a list of the *event tags* which have been associated with a particular *colour*: *tag list* ≡ (*colour,* (*signature*$_0$*, time-stamp*$_0$), (*signature*$_1$*, time-stamp*$_1$), …, (*signature*$_n$*, time-stamp*$_n$), these lists are periodically sent to the relevant global monitor so that global conditions can be assessed. How regularly this occurs depends on the feedback time for global conditions. A faster feedback will require the *event tag lists* to be sent more often.

The distributed environment which requires the monitoring service must run at least one global monitoring service component. It is naturally impossible for the global monitor to have a minimum latency for all nodes though its physical location in the network may affect its response times. If there is more than one global monitor, only one global monitor should be associated with the QoS monitoring of any particular distributed application. Without this restriction there is the possibility of data inconsistencies between global monitors which can only be resolved by establishing either a centralised information store for QoS data or

managing the information store as a distributed database. Both these solutions are unacceptable.

The monitoring service tests QoS conditions for the client distributed application. These conditions are expressions of the required QoS characteristics. Conditions may be local or global in scope. Local conditions can be tested with *event tags* collected at a single local monitor, global conditions require the collation of *event tags lists* from two or more local monitors at a global monitor. Conditions are also divided into two further categories, primary and secondary.

If the status of a condition can be ascertained from the tags collected at a single monitor then the condition is local. Local conditions can be subdivided into permanent and transient. A permanent local condition will always be a local condition regardless of how the client application is distributed. These conditions must contain event tags which originate from only one local component. A transient local condition is one in which the condition may be local dependent on how the client application is distributed. These conditions contain tags which originate from more than one component which are not necessarily co-located. If two or more components are always co-located the monitor will treat them as being one component and conditions spanning these components will be considered permanent. Local conditions are always assessed faster than global conditions.

If the status of a condition can only be ascertained from the tags collected at more than one monitor then the condition is global. Global conditions can also be subdivided into permanent and transient, though no global condition has yet been constructed that must belong to the permanent class. A transient global condition is identical to a transient local condition. The status of the condition is dependent on the distribution of the application.

A primary condition failure requires the monitor to inform the client distributed application of a QoS transgression. A primary condition may be a simple condition such as a specific latency specified to remain within an upper and lower bound, or it may be a simple condition which is augmented. Augmentation allows a primary condition to fail without informing the distributed application until the pattern of failure meets some further condition. The simplest augmentation is a counter: the primary condition is allowed to fail a number of times before the monitor informs the distributed application. Augmentation can be easily expanded to encompass a range of statistical conditions suitable for QoS.

A secondary condition is only tested in the event of an associated primary condition failure. The secondary conditions can be used to pinpoint the cause of the primary condition failure. Primary conditions generally express the user-perceived QoS requirements while the secondary conditions express the underlying application QoS requirements. Provided the user-perceived QoS requirements are met the application might not halt if an application QoS requirement is broken.

The following tables exemplify the message passing between a client *A* and a server *B*. The right arrow is a send action and the left arrow a receive action; sender and recipient are written under the arrow. The **`wait`** and **`process`** commands are place holders indicating that time is passing while either nothing is happening or there are internal events of no consequence to the external viewer.

| *A* | message | *B* |
|---|---|---|
| $\overrightarrow{AB}$ | $(data1)$ | $\overleftarrow{BA}$ |
| **`wait`** | | **`process`** |
| $\overleftarrow{AB}$ | $(data2)$ | $\overrightarrow{BA}$ |

This table gives the basic message sequence for a 1:1 interaction between a client and a server. Client *A* sends some data to a method at server *B* and receives back some different data. This constitutes a pair of action/reaction events. The $\overrightarrow{AB}$, $\overleftarrow{BA}$ events are one pair, and the $\overrightarrow{BA}$, $\overleftarrow{AB}$ are the second pair. The following tables elaborate the message sequences when the monitoring service is introduced into this client server call.

| *A* | message | *B* |
|---|---|---|
| $\overrightarrow{AM_A}$ | $(col_A, seq_A, sig_A)$ | **wait** |
| $\overrightarrow{AB}$ | $(seq_A, data1)$ | $\overleftarrow{BA}$ |
| **wait** | $(col_B, seq_A, sig_B)$ | $\overrightarrow{BM_B}$ |
| **wait** | | **process** |
| $\overleftarrow{AB}$ | $(data2)$ | $\overrightarrow{BA}$ |

This table gives the basic monitored message sequence for a 1:1 interaction. Client *A* first sends a signal to its local monitor. This must occur before the call to the server even though this is the event being monitored. In many systems of remote procedure call (RPC) *A* could not perform another action until the RPC had returned. The call to the monitor must therefore occur before the call to the server in the knowledge that the call to the server will occur without undue delay. The *data*1 is then sent to the server but included with it is the sequence number. The server includes this sequence number in the signal to its local monitor. When the *event tag lists* are compared the sequence number will allow the these events to be matched and the signatures can be compared for errors.

| *A* | message | *B* |
|---|---|---|
| $\overrightarrow{AM_A}$ | $(col_A, seq_A, sig_A)$ | **wait** |
| $\overrightarrow{AB}$ | $(seq_A, sig_A, data1)$ | $\overleftarrow{BA}$ |
| **wait** | $(col_B, seq_A, sig_{A,} sig_B)$ | $\overrightarrow{BM_B}$ |
| **wait** | | **process** |
| $\overleftarrow{AB}$ | $(data2)$ | $\overrightarrow{BA}$ |

This table gives an enhanced message sequence for a 1:1 interaction. The difference from the previous sequence is that in addition to the sequence number passing from the client *A* to the server *B* so does the signature generated at the client *A*. This allows the local monitor at the server *B* to compare the signatures for errors without the need to send the *event tag lists* to the global monitor. Although the lists still need to be compared to assess latencies this does increase the response time for some conditions, moving them from the class of global condition to transient local condition.

The following table gives a basic message sequence for an n:1 interaction. The client event *colour* is unique even among identical clients. However unless this *colour* is passed between the client and server there is no method for matching the action and reaction event.

| $A_n$ | message | $B$ |
|---|---|---|
| $A_n\overrightarrow{M}_{A_n}$ | $\left(col_{A_n}, seq_{A_n}, sig_{A_n}\right)$ | **wait** |
| $\overrightarrow{A_nB}$ | $\left(col_{A_n}, seq_{A_n}, data1\right)$ | $\overleftarrow{BA_n}$ |
| **wait** | $\left(col_B, col_{A_n}, seq_{A_n}, sig_B\right)$ | $\overrightarrow{BM}_B$ |
| **wait** | | **process** |
| $\overleftarrow{A_nB}$ | $(data2)$ | $\overrightarrow{BA_n}$ |

The case of n:m, many clients with many servers, appears problematic but if treated as the n:1 case and all servers are assigned the same colour for a particular reaction event prompted by clients the global monitor can identify the appropriate action/reaction pairs.

### *3.3 Conditions*

The conditions which the monitors evaluate are composed of a number of functions and operators which may be applied to the *event tags*. Local monitor receive the event tags directly from the application components while the global monitor receive the *event tags* indirectly as *event tag lists*. There are three primary functions which may be applied to the event tags directly, a number of meta-functions and the standard logical operators. All conditions evaluate to Boolean results, a false result indicating the condition has failed. Compound conditions are also permissible and there also the enable and disable operators which can activate and inactivate conditions dependent on other conditions.

| Primary function | Returned type | Purpose |
|---|---|---|
| $t(col, seq)$ | time | time of event |
| $s(col, seq)$ | signature | signature of event |
| $c(col)$ | integer | counter of event |

The *t* and *s* functions simply return the time and signature of a particular event respectively. The *c* function returns the number of times an event has occurred. These three functions are the fundamental elements required in this methodology. If the *seq* parameter is omitted from the *t* or *s* functions then the function is applied to all the event tags with the specified colour. This is a convenient shorthand notation. Therefore $t(col1, seq) < x$, means that the first event associated with *col*1 must occur before time *x* while $t(col1) < x$ means that all the events associated with *col*1 must occur before time *x*.

| Meta-function | Return type |
|---|---|
| $cmp(sig1, sig2)$ | Boolean |
| $truth(cond)$ | integer |

The *cmp* function is used to compare signatures and returns true if the signatures are the same. The *truth* function is used to count how often a condition is true and therefore in combination with the not operator how often a condition is false. These functions are required in conditions relating to counting errors.

| Symbol | Operator |
|---|---|
| ¬ | not |
| ∧ | and |
| ∨ | or |

These are the standard logical operators. Whether implication and equivalence are required to express a full range of conditions is not yet known. The existential and universal operators may also have benefits which further study of more complex condition statements might require. The shorthand of dropping the *sequence* number from the *t* and *s* functions provides a form of universal operator but may be removed if a more general form is required.

This is the basis for a language which can be used to express QoS characteristics. It is not complete; new functions and operators maybe suggested as more complex QoS conditions are expressed. In particular operators to flush the event counter and disabling or enabling conditions dependent on further conditions are of substantial interest. There is also the possibility of using *event tags* to start and stop *stopwatches* within the local monitors in order to improve the range and response of some condition statements.

It is not anticipated that a client distributed application will state the conditions it requires in these basic functions. Common QoS characteristics will be expressed using this language and will constitute a set of macros containing parameters which the client distributed application will instantiate. This set of macros would conform to an international standard such as the ISO QoS framework document. The next section illustrates how such macros can be constructed for the present ISO QoS framework.

### *3.4 Relationship to ISO QoS*

The ISO QoS framework document outlines fifty four QoS characteristics grouped into eight categories: time, coherence, capacity, integrity, safety, security, reliability and precedence. Many of these characteristics lack a clear definition and some appear to be non-measurable by any reasonable computational method. However the tagging methodology proposed enables some of these characteristics to be expressed as conditions to be monitored. Most of the categories have a generic characteristic which is specialised to create new characteristics. This section is directed to expressing the generic characteristic; expressing the specific characteristics is an extension of this process.

The time characteristics are the characteristics most closely related to the tagging methodology. The *data/time* characteristic is simply the *t* function as described in section 3.3, while the *time delay* characteristic is simply the difference of two *date/time* characteristics. An example of a *time delay* characteristic is:

$$\text{time delay} \equiv t(\texttt{col1}, n) - t(\texttt{col2}, n)$$

This expresses the latency between the *n*th action event recorded at `col1` and the *n*th reaction event recorded at `col2`. The *date/time* and *time delay* characteristics form the building blocks for characteristics in the other categories. The remaining time characteristics are difficult to express within the methodology although the inclusion of the *stopwatch* functionality discussed in section 3.3 may provide a method for monitoring them.

Many of the coherence characteristics require that a certain operation is applied within a given time frame. This can be measured by tagging the start of the operation with an action event and the end of the operation with the reaction event. These coherence characteristics are therefore camouflaged *time delay* characteristics. The spatially coherent characteristics, although offering symmetry (coherency in time **and** space), appear non-measurable by any reasonable method.

Many of the capacity characteristics can be measured by counting the number of action events from the start of a data transfer. One representation is:

$$\text{capacity} = \frac{c(\texttt{col}) \times m}{t(\texttt{col}, n) - t(\texttt{col}, 0)}$$

col is the colour indicating the type of a packet being sent, *m* is the size of the packet and the denominator is the time over which the entire transfer occurred. This is a very approximate characteristic relying upon a constant packet size, but does illustrate how conditions based on capacity characteristics might be constructed.

Most of the integrity characteristics are based on the accuracy characteristic. This is defined by ISO as 'the correctness of an event, a set of events, a condition, or data', and quantified as a probability

$$\text{accuracy} = \frac{c(\texttt{col2})}{c(\texttt{col1})}$$

col1 is the number of action events and col2 the number of 'correct' reaction events. This does rely on the client distributed application being able to distinguish between correct and incorrect reaction events.

The safety and security characteristics are unsuitable for this methodology. There is still some confusion in the framework document as to how these characteristics are to be defined and measured. It is unlikely that these characteristics will ever be suitable, constituting as they do a unique class.

Most of the reliability characteristics are based on the availability characteristic. This is defined provided that there is a contractual understanding of the term 'satisfactory'. The standard does however suggest that availability can be defined in terms of 'mean time between failures' (MTBF) and 'mean time to repair (MTTR):

$$\text{availability} = \frac{\text{MTBF}}{(\text{MTBF} + \text{MTTR})}$$

This can be re-expressed as:

$$\text{availability} = \frac{\sum_{n=0} t(\texttt{col1}, n) - t(\texttt{col2}, n)}{\sum_{n=0} t(\texttt{col1}, n) - t(\texttt{col2}, n) + \sum_{n=1} t(\texttt{col2}, n) - t(\texttt{col1}, n)}$$

col1 event is the resource becoming unavailable and col2 event is the resource becoming available. MTBF becomes the sum of the time that the resource is available and MTTR becomes the sum of the time that the resource is unavailable. No upper limit is given as this is constantly moving upwards. There are simpler ways of expressing availability but the above exemplifies the ease of translating a common characteristic directly into the tagging methodology.

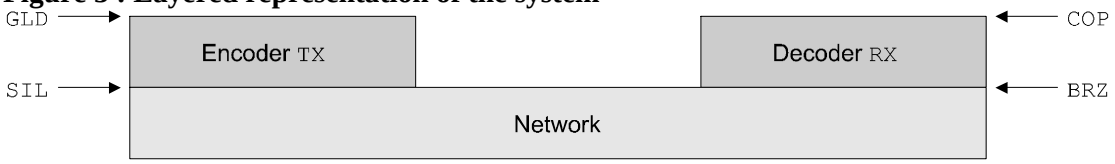
The precedence characteristics are non-measurable, conveying priority information. However these characteristics will be of use in directing the infrastructure when resource conflicts arise.

## 4. Example Monitoring Service

Given that TX is a video encoder and RX is a video decoder a simple monitoring scheme can be constructed using the proposed monitoring service. It is assumed that there is an intervening network between the encoder and decoder but no specific transfer protocol is required or expected.

The following scheme utilises four 'colours' with the identifiers; gold (GLD), silver (SIL), bronze (BRZ) and copper (COP). The gold and silver colours are associated with events at TX while the bronze and copper colours are associated with events at RX. The TX associates the gold and silver colours with the events receive frame (GLD) and transmit data (SIL). The RX associates the bronze and copper colours with the events receive data (BRZ) and display frame (COP). See Figure 3.

**Figure 3 : Layered representation of the system**




Each time one of the monitored events occurs in the distributed application a context-specific signature is generated. This signature is composed into a monitor signal and sent to the local monitor. The application does not wait for an acknowledgement and the monitor does not issue one. The monitor generates an event tag by time-stamping the signal and checks that the local conditions are satisfied. The event tag is stored in the event tag list which is periodically sent to the global monitor. A number of simple conditions are now outlined for the distributed application.

TX local primary conditions:

$$t(\mathtt{GLD},0)+\frac{n}{25}-\frac{1}{1000}\le t(\mathtt{GLD},n)\le t(\mathtt{GLD},0)+\frac{n}{25}+\frac{1}{1000}$$ rate/jitter characteristic

This TX local condition express the rate at which the frames are captured. It is expresses the expectation of the time signature on the *n*th event after the initial event. The condition also expresses an allowed jitter range. Failure of this condition would indicate the camera is failing to meet the QoS requirements.

RX local primary conditions:

$$t(\mathtt{COP}_0)+\frac{n}{25}-\frac{1}{1000}\le t(\mathtt{COP}_n)\le t(\mathtt{COP}_0)+\frac{n}{25}+\frac{1}{1000}$$ rate/jitter characteristic

The RX local condition express the rate at which the frames are displayed. It is expresses the expectation of the time signature on the *n*th event after the initial event. The condition also expresses an allowed jitter range. Failure of this condition indicates a failure of the application **but** does not indicate the location of the failure. If the TX primary condition has also failed then the failure is likely to have occurred at the camera. If this condition is intact then the secondary conditions must be examined.

Periodically the event tag list is sent to the co-ordinating monitor which assesses the global conditions by a comparison of event tags. The comparison of event tags is a nontrivial exercise requiring use of global clock values. The conditions are stated given that missing event tags are assigned an undefined time and a null signature.

TX/RX global primary conditions:

$$t(\mathtt{COP}_n)-t(\mathtt{GLD}_n)\le\frac{1}{25}$$ upper bounded latency characteristic

$$\sum_{n=0}^{n=m} cmp\left(s(\mathtt{SIL}_n),s(\mathtt{BRZ}_n)\right)\le 10$$ maximum errors allowed

The TX/RX global conditions express latency and error characteristics. The latency characteristic is expressed as the latency perceived by the user of the system, it is the time between capturing a frame and displaying a frame. The error characteristic is expressed as the number of signatures that are not consistent between the send and receive data events. This obviously includes damage and loss but not duplication.

TX local secondary conditions:

$$t(\mathtt{SIL},n)-t(\mathtt{GLD},n)\le\frac{2}{125}$$ upper bounded latency characteristic

RX local secondary conditions:

$t(\mathtt{COP}_n) - t(\mathtt{BRZ}_n) \leq \frac{1}{125}$ upper bounded latency characteristic

TX/RX global secondary condition:

$t(\mathtt{BRZ}_n) - t(\mathtt{SIL}_n) \leq \frac{2}{125}$ upper bounded latency characteristic

The choice of the secondary conditions values was made to ensure that given a primary condition failure at least one of the secondary conditions would fail[4].

This example expresses conditions for rate, jitter, latency and error. Although these characteristics are crude it is evident that the expression of QoS characteristics by tagging events is extremely powerful. In combination with a light-weight application level protocol it is possible to extend this methodology to include other scenarios.

## 5. Conclusion

At present this monitoring methodology is tailored for systems whose data is identical for the events that are being monitored. The transferred data is then used to establish the causality between the action and reaction event. This is particularly suited to monitoring network communications such as found in protocol stacks.

Future work will extend this methodology to systems in which the data is transformed between events and to causally related events between which there is no data transfer. This will naturally require the designer to embed a method of establishing the action/reaction events. It is anticipated that tools for automating this procedure will be developed, thus minimising the impact of this methodology on the designer and programmer.

## References


1. Kramer, A., *DIMMA Amber*; APM.1669.00.01: 1995, APM Ltd., Cambridge, UK.
2. Kramer, A., *The Amber Project*; APM.1686.00.01: 1996, APM Ltd., Cambridge, UK.
3. Otway, D., *DIMMA overview*; APM.1439.02: 1995, APM Ltd., Cambridge, UK.
4. *ANSAware/RT 1.0 Manual*; APM.1476.01: 1995, APM Ltd., Cambridge, UK.
5. Li, G., *ANSAware/RT 1.0: Programming and Systems Overview*; APM.1460.01: 1995, APM Ltd., Cambridge, UK.
6. Macmillan, I., *ANSAware/RT 1.1 Release Notes*; APM.1675.00.01: 1995, APM Ltd., Cambridge, UK.
7. *Information Technology - Quality of Service - Framework - Final CD.* Open Systems Interconnection, data management and Open Distributed Processing. 1995: ISO/IEC JTC1/SC 21.


[4] since $\frac{2}{125} + \frac{1}{125} + \frac{2}{125} = \frac{1}{25}$