Proceedings of Sixth ACM Symposium on Operating Systems Principles (November 1977) 93-99.

METRIC: a kernel instrumentation system for distributed environments

Gene Mcdaniel
Xerox Palo Alto Research Center (PARC)
Systems Science Laboratory
3333 Coyote Hill Road
Palo Alto, CA 94304

Extended Abstract

Metric is a distributed software measurement system that communicates measurement data over the PARC computer network, the Ethernet. Metric is used to instrument stand alone and distributed computer systems (it works in an environment of about 90 machines total and is used by about 15 machines).

The system is divided into three parts: object system probes that transmit measurement events, the accountant that receives and stores those events, and the analyst that manipulates the data for the user.

Measurement events, small packets of standardly formatted measurement data, are used in a way that emphasizes their independence, history and context in a running system. Events are not counts of some system activity, they are a mini-snapshot of the state of the system when some activity begins or ends. In this way they provide context about what is happening in the system, and a succession of events provides a rich history of what has occurred in the system under study. The contextual information intrinsic to an event supports its independence -- the event carries with it the information necessary to describe what it is all about.

Metric's robustness is a direct consequence of its simplicity. Its simple communications protocols and the independence of its parts prevent failures in the Metric system from interfering with the user's object system. Most failures in the object system are unlikely to interfere with the functioning of the Metric system. The standard format of events enables the accountant to receive events from different environments in a straightforward fashion, and makes the job of data handling easier for the analyst.

Another advantage of Metric's simplicity is its economy of use: object system probes use about 100 microseconds to transmit data to the analyst.

Object systems that use Metric continuously transmit event data. This means the event history log maintained by the accountant can be examined after particularly mysterious crashes to determine what the system had been doing lately.

The tripartite division of the analyst into the kernel, utility layer and applications layer simplifies the job of maintenance, use, and extension of the system. The kernel understands event format and acts in behalf of applications to examine data collected by the accountant. The utility layer understands global system structures and language constructs to simplify the job of data analysis and presentation. The application layer is specific code written to answer some particular questions about a system. It is usually quite small and simple.

In summary, Metric is unusual because of the way it exploits the Ethernet, its insistence on standardized measurement information, its efforts to make information intelligible to its users, and its extensibility in the face of very different user environments. The isolation of Metric's parts into different machines that communicate over the Ethernet has proven to be a very effective way of achieving a remarkably robust, low cost measurement tool. Metric's emphasis upon the context and history associated with measurements facilitates the use of measurement data.

Contents:

Introduction

Metric is a distributed software measurement system that communicates measurement data over the PARC computer network, the Ethernet [Metcalfe]. Metric is used to instrument stand alone and distributed computer systems.

Metric was developed in the environment of the Polos system [Duvall], a minicomputer network (mostly Data General Novas also connected by the manufacturer's communications device, the MCA) that emphasizes functional distribution of activities across the network (see Figure 1). The Ethernet is a 3 MBs broadcast packet switched local network that delivers packets with high probability. Software protocols are required to obtain communications with a lower error rate than that provided by the basic transport mechanism. The MCA is a standard Data General 8 MBs time division multiplexed comunications device. Errors during MCA communications indicate a serious hardware problem (or "operator intervention") as opposed to the Ethernet which employs a probabilistic approach to communications.

Polos provides a complex environment -- different machines contain tailored versions of different operating systems -- in which multiple process and multiple processor interactions confuse attempts to cause performance improvements or to find program bugs.

9

2

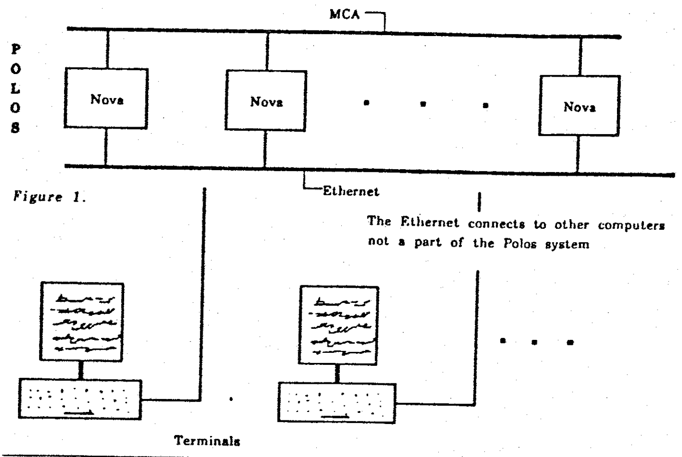

MCA
P O L O S
Nova
Nova
Nova
Ethernet
The Ethernet connects to other computers not a part of the Polos system
Terminals

Figure 1.

Metric is unusual because of the way it exploits the Ethernet, its insistence on standardized measurement information, its efforts to make information intelligible to its users, and its extensibility in the face of very different user environments. The isolation of Metric's parts into different machines that communicate over the Ethernet has proven to be a very effective way of achieving a remarkably robust, low cost measurement tool. Metric's emphasis upon the context and history associated with measurements facilitates the use of measurement data.

The first section of this paper describes the parts of the Metric system. The next two sections describe the implementation phases and the use of the Metric system.

Parts of Metric

The metric user views the world in three portions. There is a *probe* in the user's *object system*, an *accountant* that collects information from the probe, and an *analyst* that processes the information and presents it in an intelligible format. Measurement *events* are those data that the probe transmits to the accountant, and which are subsequently processed by the analyst.

In the Polos environment, the object system and the probe live in a machine that is independent of the accountant and analyst's machine. This independence plays an important role in the robustness of Metric.

The Object System Probe

The probe is the user interface to the Metric communications routines. It is a procedure call that requires a user chosen type and subtype designation (which identify the character of the event), and transmits that information along with an arbitrary amount of data to the accountant.

The Accountant

The accountant may be one or more machines or processes devoted to consuming event information from one or more object systems. The accountant acts as a recorder to which the object system continuously sends information. The accountant will accept single event packets at a time or packages of events.

The accountant may completely ignore event information, selectively keep some of the information or keep all of it. This event filtering happens by associating filtering policies with event types from various machines.

Whether or not the accountant is absent is irrelevant to the object system since probes do not customarily use acknowledgment protocols. This independence contributes to the robustness of the probe.

The accountant is a passive engine that keeps the data it accepts on a file for future reference. The accountant does as little else as possible. It is designed to be fast, simple and reliable. The only information the accountant adds to the measurement event is the identity of the event source.

Analysts

Analysts are processes devoted to analyzing and tabulating the information collected by the accountant. The Metric analyst could run on yet another machine separate from both the accountant and object system machines. An important advantage of separating the analyst from the object system is that there may be more resources available for providing users with a lucid view of the measurement data.

While analysts are inherently *ad hoc*, since they are constructed to illuminate specific attributes of an object system, the measurement event's standard format fosters code sharing among the different analysts. The implementation of an analyst may be partitioned into three sections:

1) The analyst *kernel* reads successive events from the event file and keeps track of sequencing problems (the probe automatically provides sequence numbers). This section interfaces to the event producer-consumer mechanism of Metric, and is common to all analysts.

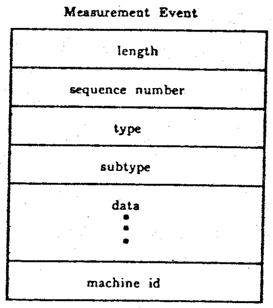
2) The analyst *utility* layer provides standard routines to print strings or numbers, read symbol tables, et cetera. This is the language and operating system specific portion of the analyst.

3) The analyst *applications* layer analyzes specific events from a specific system. This is the user portion of the analyst.

The Measurement Event

Events are small packets of state information (see Figure 2). Their standard format, especially the type and subtype designations, allows the accountant to filter unwanted events and allows different analysts to use the same code to access events stored by an accountant, or produced by different object systems.

Figure 2.



The *sequence number* is maintained by the communications code in an object system machine. It is a packet number that identifies each event sent from a particular machine. The sequence number coupled with a machine id enables the accountant and analyst to detect missing (or duplicate) events.

The *length* enables analysts to deal with events whose character or purpose they don't know by skipping them.

Event *type* is used in two ways. First, it is a broad category, a class name for experiment control. Second, it is used for resource control in the accountant: specific event types must be enabled or the accountant will ignore the event.

Event *subtypes* discriminate specific forms of events of a particular type. Subtypes are really a matter of convention for the analyst's benefit, since they are unnecessary to the producer/consumer part of Metric.

Qualities of Events

The design of events causes them to have several important qualities: they may be lost, they are independent, they provide context and history, and they contain redundant information.

There are several consequences of the fact events may be lost: It forces the analyst's implementers to design their programs to be immune to inconsistencies in the data. It facilitates lower transmission overhead by using simple transmission protocols, and it explicitly recognizes the inevitability of lost data--a fact often avoided!

When we say events are independent, we mean events are the raw information of what is happening in real time in the system being measured. The loss of a single event is unlikely to be extremely significant. By analogy, the experimenter is less concerned when he loses a single instance of a page fault statistic than when he loses the transmission of an accumulated average.

Events provide context and history with their content and sequence numbers. For example, the kind of event that might be sent when access is made to a mass storage device as part of a file I/O operation is:

probe(diskIOeventType, fileReadSubtype, fileName, TimeRequired, diskAddress, processName)
Example 1.

Events of this type provide sufficient context to determine quite a bit of information about a system. For example:

1) What is the average time to access files in the system?

2) What is the average time to access specific files?

3) How many files were accessed by a particular user process?

4) What is the average access time to files for a specific user process?

5) How much arm movement occurred on the disk?

The redundant information in a series of events reflects the duplicate information passed among the several layers of a system. Redundancy simplifies armoring the analyst against lost data and simplifies the job of data analysis.

For example, assume events from example 1, and consider investigating the effectiveness of the disk scheduling machinery. A probe in the I/O driver module may well respond to a finished request with an event like,

probe(diskIOeventType, RawDiskSubtype, TimeRequired, DiskAddress, processName)
Example 2.

While the scheduler may not know fileName, the processName and diskAddress will be the same as Example 1. The differences between the two occurences of TimeRequired will reflect queue wait time and process activation delay.

A subset of the probes used in the Polos system to investigate utiliszation of the code segment memory (CSM) further illustrates the way event data may be exploited to obtain different kinds of information:

probe(CSMType, SegmentFaultSubtype, MissingSegmentNum, EntryPoint)

probe(CSMType, AllocateSubtype, SegAddr, SegNum, SizeNeeded, SizeAllocated)

probe(CSMType, FreeSegmentSubtype, SegmentAddr, SegNum)

probe(CSMType,CompactTimeSubtype, time, spaceRetrieved)
Example 3.

These events, others like them, and diskIOevents can be used to answer questions like:

1) Which code segments fault most frequently?

2) Which code segments take the longest amount of time to swap in?

3) Which entry points are used most often?

4) Which code segments are most frequently deleted during a compaction?

5) How fragmented is the code segment memory when a compaction occurs?

The section Use of Metric describes an analyst developed to answer some of these questions.

Choosing Events

Probes are intended to be so inexpensive that they can be placed at will through a system and the experimenter/designer can choose the events to which he will listen. These guidelines have been worthwhile:

1) Place probes in the interface between layers or modules of a system.
2) Place probes at contour or block structure boundaries [Batson].
3) Place probes in the different pieces of an existing system to simulate a new, differently designed system.

Implementation phases: Line, Tree, Network

We view the orderly growth of the distributed Metric system in three phases. There is the *line*: a single object system machine and a single accountant machine. There is the *tree*: two or more object system machines that send events to a single accountant. Finally, there is the *network*: two or more trees that communicate with each other (Figure 3).

The Line

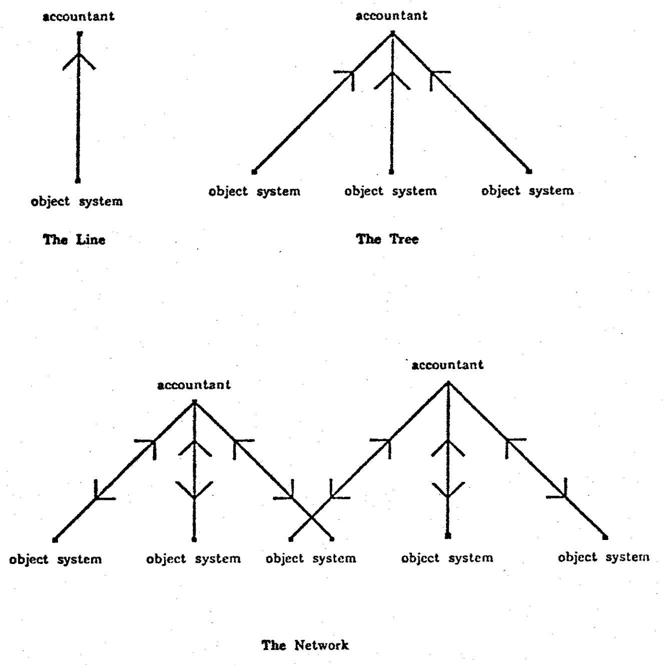
The line, the simplest form of Metric, is free of most problems associated with multiple machine interactions. It requires the basic communications machinery to work and provides a quick opportunity to obtain measurements. The line is flexible; the designer or maintainer of any stand-alone system can find a machine to run an accountant and obtain immediate information regarding the problems of his particular system.

The Tree

The tree offers the user his first opportunity to study the kinds of activities that happen in a functionally distributed system or to study the typical behavior of systems that run stand alone on numerous machines. For example, answers to questions like:

What are the typical compile errors in language X?
What are the most common editing operations?
How much idle time is there across the whole network?

can be collected with greater ease than if individual machines had to generate and store those sorts of statistics. The tree also conserves resources by allowing a single accountant machine to collect events from several object systems. Unfortunately the



Note: In the line, information flows from the object system to the accountant.
In a simple tree the information from several object systems collects at an accountant
In a more complicated version of the tree and in the network, information flows
both directions.

(Figure 3)

tree analyst has more difficulties with sequencing and discrimination of events.

The Network

Our ideas about networks of metric are conjectural since our experience is limited to the line and tree architectures.

Resource sharing is the principal attraction of the network version of Metric since the network offers no functional additions to the capabilities offered by tree Metric. The network offers the opportunity for object systems to broadcast a request for an accountant and then to select the machine least loaded [Farber, Thomas] as the one to which it will send its measurement events.

The network architecture will cause data to be distributed across the network. The mechanism to deal with such data is complex [Cosell]. Unfortunately, these improvements presume a fairly sophisticated Metric system or supporting file system--and sophistication breeds robust bugs before it breeds robust systems.

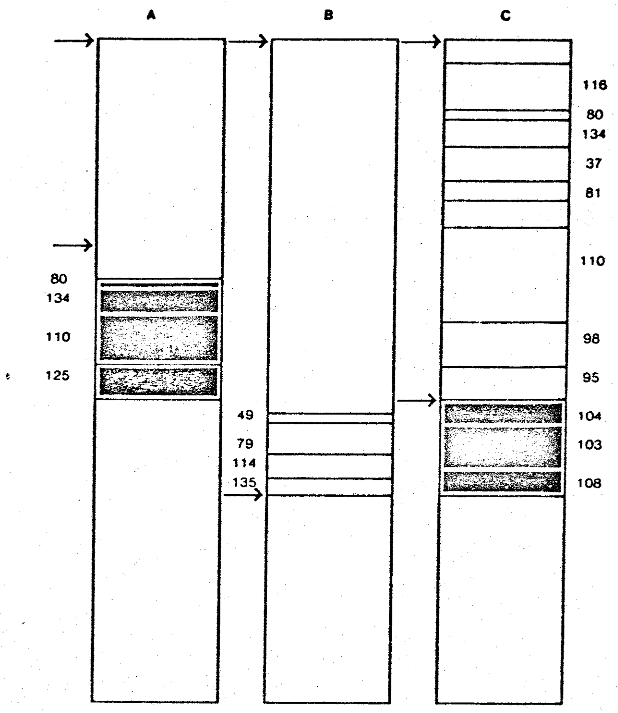
Use of Metric

Using Metric we have adopted an iterative approach to system optimization where we obtain specific information to back various implementation decisions.

Experimenting with Code Swapping: the Line

The first analyst was like a *microscope*. It provided a detailed history of the system activity and a simple statistical summary of its characteristics. We were studying code swapping characteristics of the multiple process environment in the Polos system.

The kind of thrashing indicated by those histories and statistics prompted the development of a Smalltalk [LRG] analyst that made a *movie* illustrating code swapping behavior (see Figure 4)--our *telescopic* view of the system. Such visual cueing is very important because it provides a qualitative understanding not obtained by looking at masses of system statistics [Swinehart; Wegner]. Results from the Smalltalk analyst suggested that an LRU algorithm [Coffman] or a code restructuring algorithm [Ferrari] should be adopted to improve system performance.



The graphics analyst cycles through three views of the memory zone dedicated to code segments. The top most arrow represents the current top of the code zone. The bottom arrow is the current top of the code stack. Dark boxes represent allocated code segments. As the designer watches the program run he can see holes developing in the zone, he can watch compactions, and he can see the code segments that have been deallocated recently. The analyst is currently working on view A. Notice that segment 80 was freed in view C and recently reallocated in view A.

Figure 4.

A different version of the analyst that gave symbolic information and an improved statistical profile of code swapping activity, another microscope, was developed. It suggested that the cost of code compaction in our system made an LRU algorithm unfeasible. Since then we have performed several experiments by applying code restructuring and by modifying the compaction routines. We used the analyst to check on our results.

Experimenting with Machine Interactions: The Tree

The tree structure was developed to maintain more information regarding the system wide interactions between a centralized file server and its numerous users. We have used the tree strictly to merge common event types as opposed to correlating different event types in different machines with a common activity.

Discussion

This section discusses the architecture, implementation and use of Metric. The robustness of Metric as a function of machine isolation and the economy associated with Metric's approach to communications are particularly interesting. The careful layering of the analyst has proven to be very worthwhile.

Architecture: Robustness

We want the accountant to be robust and to leave object systems unharmed in the event of an accountant failure. The acknowledgment-free probe to accountant interactions expedite this goal. Little the object system does can harm the accountant and vice versa. Failures can occur in three places:

> 1) At the worst, the object system may spew many events at the accountant. This causes excessive use of the accountant's resources and the possible degradation of the communications medium. While we have never experienced object system logorrhea, it could be an odious problem.
>
> 2) The accountant can die without affecting the object system since our implementation does not require communication between the accountant and the object system.
>
> 3) While the communications medium can die, the unidirectional, "no acknowledgments" type transmission between the object system and the accountant prevents this from harming the object system.

The criterion that Metric bugs must not contaminate the object system is easily met since only the probe's communications routines need be debugged. Those routines are small given the modest communications goals.

The analyst's separation from the rest of Metric prevents any of its bugs from directly affecting the accountant or the object system.

Architecture: Communication

Metric is the child of its environment: the design depends upon the availability of an inexpensive, high bandwidth communications line, the Ethernet.

Metric should not overload the Ethernet. Our computations show that 80 machines could send about 75 events per second averaging 14 bytes per event without seriously clogging the Ethernet.

Even so, ethernet congestion can become an issue. Enough machines spewing events on the Ethernet could result in such congestion that one of several alternatives may become necessary:

> 1) Events per second per machine can be limited by administrative policy for systems not being run in "debug" or "experiment" mode.
>
> 2) Machines or systems with the right to send measurement events may be limited by administrative policy.
>
> 3) A traffic controller can monitor Ethernet usage and ask particularly verbose machines to cease sending events. This course will complicate the probe portion of Metric.
>
> 4) A private communications line could be provided for high bandwidth event flow. This is easy and inexpensive in the Ethernet world.

Our attitude toward the unreliability of events permits *low* overhead. Reliable packet transfer protocols require maintaining state information, an awareness of time outs and sending acknowledgments [Akkoyunlu; Cerf; Metcalfe]. These requirements are not compatible with Metric's low overhead ideals. Metric loses about .5% of the events transmitted to it because of the simple protocols.

For those events that must be transmitted more reliably, extra steps may be taken:

> 1) The information can be transmitted redundantly.
>
> 2) The information can be transmitted to more than one accountant.
>
> 3) The information can be stored locally and retrieved manually.
>
> 4) The accountant can be expanded to allow a protocol socket where appropriate conventions are followed to enhance the probability of reliable transmission.

Architecture: The Producer/Consumer Relationship

The producer/consumer division in the Metric system can be implemented using a small, private Ethernet, the MCA, another communications device, or another process within the same machine as the object system. In the latter case, the process would buffer events and save them on some available mass storage device for future reference.

> The character of the event producer is independent of the accountant itself. The event producer may be any machine running any language or operating system. The standard event format allows any system to participate at a minimal cost to its measurers (who must construct an analyst that suits their needs).

Implementation: Overhead Costs

We have found there is small cost to using Metric, that costs increase uniformly with the quantity of measurements taken, and that there are no sudden peak costs. Metric is cheap to use.

The time to transmit events is small--on the order of 100 microseconds in our system (1 MIP processor). The code space for the Ethernet drivers and the probe() routine is about 190 sixteen bit words--about 20 words of machine language and 170 words of resident Bcpl code, the language we use in Polos object systems [Curry]. Each probe() call takes about 10 instructions of procedure call overhead.

The economy of Metric is highly dependent upon the characteristics of the Ethernet. The Ethernet is easy to drive and using it does not significantly degrade the performance of object systems.

Implementation: Controlling Resource Consumption

To conserve mass storage utilization, the accountant understands the idea of filtering events. When an event arrives, the accountant compares the event type against a list of acceptable event types for that machine. If there is no match, the event is thrown away.

In any system, the accountant will eventually exhaust the available mass storage. To preserve the history inherent in the event stream across user "runs", the accountant must avoid prematurely destroying old data. It may implement a circular buffer arrangement or "double buffer" its events into different files. The circular buffer arrangement is complicated since events are different sizes. Switching back and forth between two or more event files guarantees there will be at least one file of event history.

With increasing usage and limited storage, the facility to maintain "counters" may have to be added to the accountant. These counters would be modified in core each time an increment or decrement was received from an object system. Counters can conserve event storage space at the expense of accountant cpu time and complexity. They also reflect a lapse in the compartmentalization between the accountant and the analyst.

Use: Debugging

Our contention has been that measurement information should flow continuously from a system--for debugging as well as performance and system analysis [Lauesen]. Continuous debugging traces are useful since the historical context provided by the accountant points to the causes of mysterious "cosmic ray" bugs that are not repeatable.

Use: The Analyst.

There are several problems associated with the analyst that bear further discussion: duplicate data structures, verifying the results, and good human engineering.

The distributed nature of the Metric system naturally causes information to exist in two places and causes updating problems.

> For example, there are declarations that define event *type* and *subtype* values. This information must be copied from the object system environment to the analyst's environment. If changes in the object system are not be updated in the accountant's data structures, chaos results when data is analyzed. This problem is exacerbated by the use of different language systems to implement the object system and the analyst.

The unanticipated results of systems measurements further complicates the problem of verifying the results: When the experimenter obtains unexpected results he must decide if he has learned something new about his system, or if he has misused his measurement tool, or if that tool has somehow failed.

Given that the probe-accountant-analyst ensemble works, care must be taken to present the pertinent information rather than hide it. Filtering out masses of data or presenting specific relationships immensely eased the use of the system and facilitated understanding the object system.

Extending the kernel: Higher Level Language Instrumentation

Now we discuss the suitability of Metric as a kernel for further extension, and use the work of [Mac Ewen] to develop criteria for a distributed instrumentation system in a heterogeneous network:

> 1) The system should be easy to use.
> 2) The system should provide a uniform context for talking about measurements.
> 3) The system must be efficient.

We want to obtain measurements from a system without recompiling or reloading it. In Metric, event types can be enabled at the accountant independent of the state of the object system. This is only an advantage if the object system probe already sends the needed events. Where new information is desired, another mechanism must be added. This implies some sort of code patching mechanism [Deutsch] or redefinition mechanism [Mac Ewen]. The Metric kernel completely ignores this issue.

A patching facility is an attribute of the language support facilities in the object system. If such facilities exist the Metric routines can be used with them.

We can afford to "take down" a machine to add measurement facilities since it takes us as little as five minutes to implement new measurements. In a production-oriented environment, even five minutes of down time may be unacceptable. Then the ability to add probes to a constructed or running system is more important.

The names and structures (the context) visible in the object system where the probe is used should be visible to the Metric user. Providing such uniform context is difficult when the probe and analyst may be implemented on different processors in entirely different environments.

For example, the analyst should be able to help users deal with probe data from an APL machine, a machine dedicated to information retrieval, and a development machine being used by systems programmers.

While we agree with the importance of allowing the Metric user to see the same context as his object system provides, we feel this is inappropriate in the kernel. The fundamental context in Metric is the machine.

> Consider the case of a process-based system. Process identification can be provided in another layer in the object system by encapsulating the Metric version of the probe() routine with a routine that always includes process information with the data. A part of the analyst utility layer will know about processes, so that different applications of the Metric system that use the same operating system can use the same code.
>
> The existence of heterogeneous system environments prevents the Metric kernel from knowing very much about the systems it measures.

Another crucial issue is whether the sorts of tools provided by an object system can be employed outside the running environment of the object system. An analyst for a particular object system may well be run on a different processor and in a different environment from the object system. That analyst's implementer will want to access the tools and context of the language and object system being analyzed. Metric doesn't address this issue.

> An important part of the development of the two Bcpl analysts was the design and construction of naming facilities that enable the Metric user to get names rather than addresses. These facilites will work for any of our Bcpl systems that Metric measures; *we view them as crucial facilities.*

As mentioned above, we believe the simple interface to Metric in the object system costs very little in terms of object system machine cycles or code space. Most extensions are present in the analyst where they don't impact object system performance. The object system extensions come in two flavors: user support and run time context identification.

User support consists of things such as adding or deleting measurements, or displaying measurement status upon user demand. These features tend to be things that can be run in the background, swapped in, or otherwise kept at a low cost profile.

The price of context identification depends upon the complexity of the object system in question. When complex context affiliations last for a long time, they can be associated with an abbreviation that accompanies the measurement event. This can be done in the same fashion of adding process context information that we discussed above. Short term, complex affiliations require more data to be transmitted in the measurement event.

Conclusions

Metric has obtained the goal of low cost measurements. The infrequent loss of events, about .5%, has not proven to be a problem in our experiments. The accountant works in a fairly large and complex environment (one computer network of about 15 machines and another of about 70 machines).

The carefully enforced separation of responsibilities into the probe, accountant and analyst along with a standard data packet format has greatly simplified the production and processing of measurements.

Metric is easy use and easy to change. The layering of the analyst into an event consumer layer, a language and operating system layer, and a user applications layer, has simplified the construction of analysts. The standard event format ties these diverse pieces together. We have successfully spanned a great deal of diverse data in ways that have aided our understanding of the systems we have built and extended.

Acknowledgments. I want to thank Dan Swinehart and Howard Sturgis for sundry encouragements and ideas. Thanks to A. Goldberg, D. Swinehart, M. Schroeder, D. Boggs, J. Shoch, and W. Newman for help with debugging the paper.

References

[Akkoyunlu] E.A. Akkoyunlu, K.E.Kanadham, R.V. Huber, Some Constraints and Tradeoffs in the Design of Network Communications, *Proceedings of the Fifth Symposium on Operating Systems Principles*, 1975.

[Batson] A.P. Batson, R.E. Brundage, Segment Sizes and Lifetimes in Algol 60 Programs, *CACM*, 20,1, 1977.

[Cerf] V.G. Cerf, R.E. Kahn, A Protocol for Packet Network Intercommunication, *IEEE Transactions on Computers*, Volume Com-22, Number 5, May 1974.

[Coffman] E.G. Coffman, Jr, P.J. Denning, *Operating Systems Theory*, Prentice-Hall, 1973.

[Cosell] B.P. Cosell, P.R. Johnson, J.H. Malman, R.E. Schantz, J. Sussman, R.H. Thomas, and D.C. Walden, An Operational System for Computer Resource Sharing, *Proceedings of the Fifth Symposium on Operating Systems Principles*, 1975.

[Curry] J. Curry, *A BCPL Manual*, CSL, Xerox PARC 1974.

[Deutsch] L.P. Deutsch, C. Grant, A Flexible Measurement Tool for Software Systems, *IFIP 74, Software*.

[Duvall] W.S.Duvall, *POGOS An Operating System for a Network of Small Machines*, SSL 76-6, Xerox PARC, June 1976.

[Farber] D.J. Farber, et. al., The Distributed Computing System, *Proceedings of the Seventh Annual IEEE Computer Society International Conference*, San Francisco, February, 1973.

[Ferrari] D. Ferrari, Improving Locality by Critical Working Sets, *CACM*, 17,11, 1974.

[Lauesen] S. Lauesen, A Large Semaphore Based Operating System, *CACM*, 18,7,1975.

[LRG] Learning Research Group, *Personal Dynamic Media*, SSL 76-1, Xerox PARC, Palo Alto, CA, March, 1976.

[Mac Ewen] G. Mac Ewen, On Instrumentation Facilites in Programming Languages, *IFIP 74, Software*.

[Metcalfe] R.M.Metcalfe, D.R.Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks, CACM*, 19,7,1976.

[Swinehart] D.C. Swinehart, Copilot: a Multiple Process Approach to Interactive Programming Systems, Stanford PhD Thesis, Stanford University, 1974.

[Thomas] R.H.Thomas, A Resource Sharing Executive for the Arpanet, *Proceedings of the National Computer Conference*, 1973.

[Wegner] P. Wegner, Data Structure Models for Programming Languages, *Symposium on Data Structures in Programming Languages*, 1971.