

Software Probes and a Self-testing System - for Failure Detection and Diagnosis ¹

Ram Chillarege

T.J. Watson Research Center, Yorktown Hts., NY
(914) 784 7375, Fax: 784 6201, ramchill@watson.ibm.com

Abstract

A key problem in today's complex software systems is software failure detection and isolation. Given that most software failures are only partial and if efficiently diagnosed, isolated and recovered, they could avert a total outage. The *probe* detects failed software components in a running software system by requesting service, or a certain level of service, from a set of functions, modules and/or subsystems (target) and checking the response to the request. The objective is to localize the failure only up to the level of a target, however, achieve a high degree of efficiency and confidence in the process. Targets can be identified at different levels or layers in the software, the choice based on the granularity of fault detection that is desired, taken in consideration with the level at which recovery can be implemented. The implementation of the probe system is made self testing against any single failure in its operational components, using the idea of a *null probe*. The probe system has been designed taking advantage of the latency characteristics of errors to provide a low-overhead mechanism. The ideas are implementable in either a single or multiple computer system.

Keywords: Software Errors, Failure, Detection, Isolation, Diagnosis,

1 Problem Definition

Software failure has continued to be a major concern in system reliability as it can cause loss of availability in either the entire system or specific subsystems. As the installed software in systems get larger and more complex, so have some of the failure modes experienced in them [8][9] [6]. There are two important issues to be noted with software failures: One: Not all software errors, and failures they cause, catastrophic to the system. Two: software errors have a very large error-latency. These issues have been revealed through several publications on error characterization in the past few years and we will discuss them, only briefly, to motivate the ideas in this paper.

In a large operating system it is possible for certain parts of the system to be damaged and while these services are unavailable to the user, other services remain un-hampered. In fact, it is common practice in production systems to delay the "IPL or reboot" to avoid downtimes during the prime shifts when such partial failures are experienced. On the other hand, several studies from the 80's showed that system failure tended to occur during high loads [5] [1] and measurements showed that indeed large error latencies [3] could exist. A detailed study through fault-injection on an MVS system with an IMS (database) workload revealed several important answers [2]. The experiment emulated a high impact software error and via acceleration showed that while only around 15% of the errors caused a complete failure, at least a fifth caused potential hazards and a third caused only partial failures. Thus, partial failures and latent errors dominated the distribution.

Integrating these facts a clear direction emerges:

- The largest bulk of the failures are only partial failures, which do not (individually) cause a total system outage. Therefore, when identified quickly and recovered they will not only improve overall availability but may also avert a total outage.

¹ Cleared for publication

- Latent errors are the next largest fraction are like a time bomb set to go off. They can accumulate in number and surface together by a common trigger (e.g. shift in workload). Sweeping the latent errors (one by one) and recovering the services they would impact is critical to avert a large outage.

This paper proposes the idea of a software probe to help detect failed software services and latent errors efficiently. The implementation of the probe system provides an on-line mechanism for periodic testing and diagnosis. The design point takes advantage of the knowledge that the error-latency are large in operating systems (several minutes once, past the fast fail), to provide a low-overhead technique. The implementation is self-testing and is extensible to multi-systems.

2 Probe Concept

A probe is defined on a service delivered by the software and is not meant to be a rigorous test of function. The thesis is that an extensive test of function is not necessary to identify failure, and most of the time a simple test of services will suffice for a particular component or sub-system.

For this paper, we term *targets* as a set of services that are abstracted from the functionality of the entire software system. Thus, a target can refer to a service provided by several modules within a component or subsystem, or may refer to one of the several services provided by a module. The key is that the targets are defined on the service, not the logic of implementation. To provide the service some or all parts of the software in the target need to be exercised. In the process of providing the service, they can influence other service providers because of dependencies that may exist among all the targets in a system.

A probe, request services from a target or a change in the level of services provided by the target. If the service is provided in a timely fashion, then the probe labels the target as **healthy**, if not **failed**. The probe is therefore specific to a target and is written specifically for the services provided by the target containing all the necessary information to check the service against the current or a nominal index.

However, the probe is not defined on the specific logic that is used to implement a block of code within a module. Thus, it is not like the acceptance test [7] to check correctness of operation of the code just executed. It is intended to be a higher level service oriented request that creates or decreases work in the system. The objective here is to localize the failure only up to the level of a target, however, and achieve a high degree of efficiency and confidence in the process.

Targets are chosen such that they represent a collection of functions that can be defined by a service level input/output (I/O) specification. Clearly, targets can be identified at different levels or layers in the software. The choice of a level is based on the granularity of fault detection and isolation that is desired, taken in consideration with the level at which recovery can be implemented and the overhead of periodic testing. A hierarchical organization of targets allows for efficient periodic testing and good diagnostic capability.

A simple examples illustrate the basic concept of the probe. Consider first a telecommunications subsystem which is part of a transaction system. In fault injection experiments [2] on IBM's Information Management System/Virtual Storage (IMS/VS database product it was shown that there exist failure modes in the telecommunications system where the failure does not affect either the IMS application nor alert the operator. However, the fault does not permit any new sessions to be set up. A probe that exercises the communication subsystem function of session management will detect and isolate the failure.

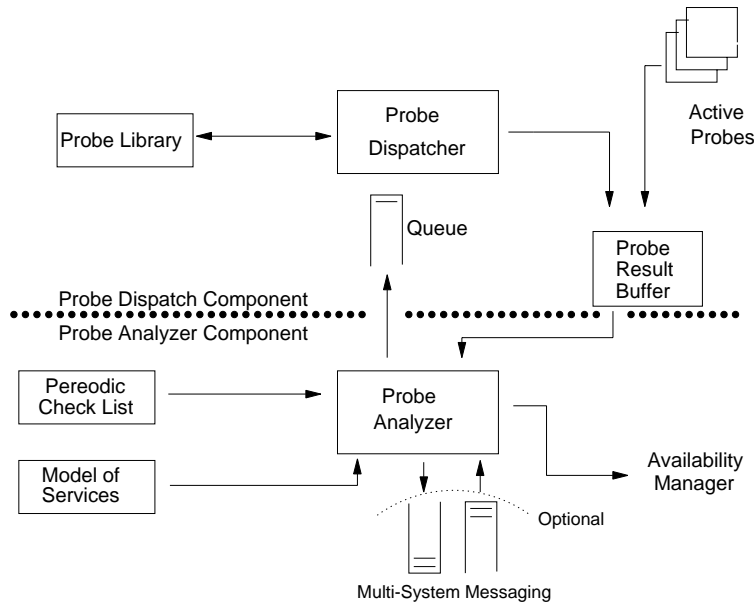


Figure 1: The Probe System

3 The Probe System

The operation of a system with probes can be viewed as partitioned into two operational components that communicate with each other. Figure 1 shows this organization. The probe dispatch component dispatches probes and records the inference on the target's health in a probe result buffer (PRB). This component includes a probe dispatcher which accesses a probe library in response to a queue of probe requests. The dispatched probes are active probes until a result is returned from their respective targets or until a predetermined time has elapsed. The results of the active probes are posted in the probe result buffer.

The other component of the probe system is the probe analyzer component. The probe analyzer component includes a probe analyzer which analyzes the results in the probe result buffer to infer the health of the overall system. The analyzer is, in turn, driven by a periodic check list of the probes to be performed. In addition, the analyzer receives as an input a model of the services the software modules are expected to perform. This model is used by the analyzer to evaluate the results in the probe result buffer. Based on the analyses performed, the analyzer provides an output to the availability manager to indicate the failure of software services.

In operation, a probe is initiated by the analyzer queuing a request in queue to the dispatcher. The dispatcher starts the probe and makes an entry in the probe result buffer, as shown in more detail in Figure 2. This entry contains a probe-id (probe identification), a unique sequence number, a time stamp and the maximum time allowed for the probe to return with an inference. This record is completed by the probe by labeling the target as healthy or as failed. The dispatcher periodically inspects the probe result buffer to check if the probes that were dispatched updated their respective status fields. If a probe does not return within the maximum time for the probe, the dispatcher writes a lost field in the record. The probe could return late and update the status even though it has been marked lost. However, the sequence of events will be visible to the probe analyzer since each update to the

← Entry by Dispatcher →				← Entry by Analyzer →		← Entry by Dispatcher →	
Probe ID	Seq No	Time	Max Tm	Status	Time	Lost?	Time
Prb 8	001	00.01	00.45	healthy	00.23		
Prb 2	002	01.00	00.45	healthy	01.27		
Prb 6	003	01.22	00.30	healthy	01.35		
Prb 8	004	01.57	00.45	failed	02.25		
Prb 6	005	02.30	00.20			lost	3.00
Prb 8	006	03.45	00.45	healthy	03.87		
Null Prb	007	04.15	00.45			lost	4.80

Figure 2: The Probe Result Buffer

probe result buffer leaves a time-stamp. A lost probe indicates a failed probe or a failed target. Lost probes can be further differentiated by the analyzer.

The probe analyzer drives the probe dispatcher from a periodic checklist and uses the probe result buffer to determine failed services. Since the analyzer is the component that drives the dispatcher, it has the capability to explore related targets that may be affected due to a failure and determine the extent of damage. This is an important ability which will help guide the recovery process with the help of an availability manager. This enhanced ability of an analyzer needs logic that can be based on some model of the system. Such a model should contain a set of dependencies of services with associated targets and their probe- ids. The degree of detail in such models is dependent on the recovery capabilities that are available through the availability manager.

3.1 Dispatcher Component

Figures 3(A&B) are flow charts illustrating the dispatcher logic. The dispatcher logic performs two functions, that of dispatching probes in response to probe requests in the queue and that of managing the data entries into the probe result buffer. These two functions are represented by two threads. The first thread, shown in Figure 3(A) illustrates the function of dispatching probes and begins by inspecting the queue. A test is made to determine if the queue is empty and, if it is, the process loops back, with a delay, to await a probe request in the queue. When the queue is not empty, the usual case, the top entry is picked up . Based on the specific probe request, a probe is retrieved from the probe library. A test is made to determine if the probe is full. If so, the process loops back, with a delay; otherwise, the retrieved probe is dispatched. Before updating the probe result buffer (PRB), a lock is acquired, since the buffer is shared by more than one thread. The status of the dispatched probe is updated and provided with a time stamp, this information being needed for maintaining the probe result buffer. The probe lock is released before control is returned.

First Thread

Second Thread

Figure 3: Dispatcher Logic

Figure 4: Analyzer Logic - First Thread

The second thread, having to do with managing the data entries in the probe result buffer, is shown in Figure 3 and begins by accessing top of the buffer. A test is then made to determine if the top entry in the buffer was the last entry and, if it was, the process loops back, with a delay, before again attempting to process entries in the buffer. Assuming that the top entry was not the last entry, then a further test is made to determine if the entry has the status of either healthy or failed. See Figure 2 for examples of such entries. If the entry has the status of either healthy or failed, then the next entry is accessed, and the tests of decision blocks repeated. If the entry does not have the status of healthy or failed, a test is made to determine if the time has passed for receiving a response to the dispatched probe. If not, the process accesses the next entry via; otherwise, a probe lock is obtained. After the lock is granted, the entry is marked as lost and the lock released before the next entry is accessed. The status of lost is indicated for one of the entries of the probe result buffer in Figure 2. A lost probe indicates either a failed probe or a failed target, and this is differentiated by the probe analyzer.

3.2 Analyzer Component

The analyzer, like the dispatcher, has two threads. The first, and simplest, is loading the probe queue to initiate the dispatch of specific software probes. The second thread analyzes the data entered into the probe result buffer as a result of the probes. Figure 4 illustrates the first thread, and Figure 5, the second. The process begins by accessing the next probe request from the periodic check list. The accessed probe is given a sequence number. A test is made to determine if the queue is full. If so, the process loops back, with a delay; otherwise, the sequence number is entered into the probe result buffer by the probe dispatcher, and this number is used by the probe analyzer to indicate the precedence of the probe. The probe request is then queued in the probe request queue. At this point, the process loops back, with a delay, to access the next probe request.

Referring now to Figure 5, the process of accessing and processing data entries in the probe result buffer begins by initializing the index number to zero followed, by incrementing the index number. A test is then made to determine if the last entry has been processed. If so, a return to the main program shown in Figure 4 is made; otherwise, a further test is made to determine if the entry has the status of healthy. A healthy status is entered whenever the software module called by the probe returns the expected result within an expected time period. If the status is healthy, the analyzer requests a probe lock, and when the lock is granted, the entry is deleted from the probe buffer. The process then loops back where the index number is incremented to access the next entry in the buffer.

If the status of the entry is not healthy, a further test is made to determine if the status of the entry is failed. If not, an additional test is made to determine if the status is lost. If not lost, the process loops back where the index number is again incremented to access the next entry.

If the status of the entry is failed, the model of the service for that probe is accessed from the model of services library to determine if the target of the probe has any children. Figure 6 illustrates a sample hierarchy of software of targets determined by the dependency relationships and also the degree of diagnostic ability required. Only the level of nodes indicated by the dotted line is subject to the periodic checks stored in check lists. If one of those nodes (i.e., target) fails, then the analyzer looks for children of that node. If children are found, these are marked for probe requests. The analysis process continues with probe requests added to the probe queue and posted to the buffer until nodes with no children are reached. In this way, the specific node which has failed can be identified. This identification is useful for diagnostic purposes, but it can also be used by the analyzer to mark a particular module as being unavailable. In this way, the reliability of the system is maintained, although with degraded performance for some functions.

Returning to Figure 5, the failed node is tested for children, and if there are children, the probe analyzer accesses probe requests for each of the children before the process returns to the main program shown in Figure 4. If there are no children, the probe analyzer provides an alert output. A further test is made to determine if the failed module is itself a child of a higher level module either at or below the dotted line in Figure 6. If so, an output is made to the availability manager which marks the module as being unavailable but allows processing of other children of the higher level module which have not failed. If not a child or after posting to the availability manager, the process loops back to access the next entry in buffer.

Now, if the status of the probe is lost, a further test is made to determine if the probe was a null probe. If not, control goes to processing as if the status were failed. On the other hand, if the probe was a null probe, then the process goes directly to an alert.

3.3 Self-Testing using the Null Probe

The purpose of the null probe is to test the probe dispatcher as the target using the probe paradigm. This is accomplished very effectively by having a probe that has no target which also does not return a health status to the probe result buffer. The effect of requesting a null probe by the analyzer is that the probe dispatcher treats it as any other probe and performs the function of dispatching it and making an entry in the probe result buffer. However, the null probe by definition does not return any status. This will cause a lost status to be posted by the dispatcher in the probe result buffer after the maximum time for the probe elapses.

Figure 5: Analyzer Logic - Second Thread

analyzer) is healthy. This is useful in a multiple machine configuration where each machine can have its own probe system.

3.4 Extension to Multi-System Probes

It is possible to extend the probe concept and the probe system to a multi-system environment. There are several ways in which to organize the communication and sharing of functions between the dispatch and analyzer functions for probes from other systems. For the purpose of this short paper, only a few of the major design choices are high-lighted.

The major design point is determined by whether there is shared virtual storage between the multiple systems or not. This influences whether the probe-result buffers are made visible to the other systems or only the relevant contents are made available through message passing. This is primarily driven by a performance decision but has serious implications on the semantics of how remote-probes are handled.

A implementation for multi-system extension follows treating each probe system independently, providing the capability to dispatch probes on the other systems, and return the results of probes to the systems that originated the request. The only major changes necessary are an in and out queuing capability to the Probe Analyzer (shown as optional in Figure 1) and an additional column in the Probe Result Buffer for system ID (not shown).

Multi-system messaging (which is assumed as a base service) then is used to queue requests to other systems and receive the results of the probes. One of the approaches will be to allow notification of failure to an availability manager only through the originating Probe Analyzer since, it can then determine through additional diagnostics when to make a specific alert. Clearly, there are several design trade-offs that are largely determined on the functionality that is provided in the base messaging services. For example, if the messaging services provides ordering and guaranteed delivery then, the semantics of the inter-system probes are different from the case where the messaging services do not guarantee delivery.

It should also be noted that a null probe dispatched on a remote system provides a self test, not only for the other probe system, but also for the communications or messaging system that tie the two probe systems together. Depending on the multi-system messaging capability (ordering, guaranteed delivery, etc.) the null probe paradigm can be extended to provide for self testing the probe system across systems.

4 Summary

This paper addresses a key problem related to detection and diagnosis of software failures. As software systems have become more complex and the failures tend to dominate there is a need for faster and better error and failure detection mechanisms. To address the above problem, this paper defines the Probe and provides the Probes system which is self testing, and can be implemented on either single or multiple systems.

Specifically:

- The Probe, is build on the observation that most software failures do not cause a total outage and also have a large latency. Software failure also are associated with hard diagnostic problems.
- Defines the probe on the service provided by set of modules (target). The probe will label the target as healthy or failed depending on the responsiveness to the service requested on the target. Probes are not an

extensive test of function or logic and thereby achieve a high degree of efficiency,.

- A Probe System composed of a dispatcher and analyzer component is described that can provide online testing and diagnosis. The analyzer can issue probes from a periodic list extracted from a hierarchical organized targets, providing diagnostic capability upon a failed target.
- A Null Probe is defined that does nothing, returning a lost status, and thereby testing the dispatcher and the analyzer components making the Probe System self testing.
- The Probe paradigm and the Probe System can be extended to a multi-system environment providing for global monitoring and diagnosis.

References

- [1] CASTILLO, X., AND SIEWIOREK, D. P. Workload, performance and reliability of digital computing systems. *Digest FTCS-11* (1981).
- [2] CHILLAREGE, R., AND BOWEN, N. S. Understanding large system failure - a fault injection experiment. *Digest 19th Fault-Tolerant Computing Symposium* (1989), 356–363.
- [3] CHILLAREGE, R., AND IYER, R. K. Measurement based analysis of error latency. *IEEE Transactions on Computers* 36, 5 (May 1987), 529–537.
- [4] CHILLAREGE, R., RAY, B., GARRIGAN, A., AND RUTH, D. The recreate problem in software failure. *IBM Research Technical Report* (November 1992).
- [5] IYER, R. K., BUTNER, S. E., AND MCCLUSKEY, E. J. A statistical failure/load relationship: Results of a multi-computer. *IEEE Transactions on Computers* 31, 7 (July 1982), 697–706.
- [6] LEVENDEL, Y. Defects and reliability analysis of large software systems: Field. *Digest 19th Fault-Tolerant Computing Symposium* (June 1989), 238–243.
- [7] RANDELL, B. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering* 1 (1975), 222–232.
- [8] SULLIVAN, M., AND CHILLAREGE, R. Software defects and their impact on system availability - a study of field failures in operating systems. *Digest 21st Fault-Tolerant Computing Symposium* (June 1991).
- [9] SULLIVAN, M., AND CHILLAREGE, R. A comparison of software defects in database management systems and operating systems. *Digest 21st Fault-Tolerant Computing Symposium* (July 1992), 475–484.