Automating Software Failure Reporting

BRENDAN MURPHY, MICROSOFT RESEARCH

There are many ways to measure quality before and after software is released. For commercial and internal-use-only products, the most important measurement is the user's perception of product quality. Unfortunately, perception is difficult to measure, so companies attempt to quantify it through customer satisfaction surveys and failure/behavioral data collected from its customer base. This article focuses on the problems of capturing failure data from customer sites. To explore the pertinent issues I rely on experience gained from collecting failure data from Windows XP systems, but the problems you are likely to face when developing internal (noncommercial) software should not be dissimilar.

A LITTLE HISTORICAL PERSPECTIVE

Traditionally, computer companies collected failure data through their customers' or their own service arm, manually submitting bug reports. Back in the 1970s and 1980s, a number of computer companies (IBM, Tandem, Digital, etc.) began to service their customers' computers through electronic communication (usually a secure telephone link). It was a natural progression to automate the collection of failure data: whenever the computer/application

crashed, its failure data was automatically collected and sent back to the manufacturer, where it was forwarded to the engineering department for analysis.

The initial focus of these processes was to address failures that occurred because the product did not perform its "advertised" functionality (i.e., system crashes resulting from software bugs). As Jim Gray, now head of Microsoft Bay Area Research Center, found in his analysis of the failures occurring on Tandem systems, however, a large proportion of customer failures occur as a result of user actions, often categorized as HCI (human-computer interaction) failures (Census of Tandem Systems Availability between 1985 and 1990, *IEEE Transactions on Reliability* 39,4: 409-418). This is in spite of the Tandem systems being managed and serviced by highly trained personnel. The root causes of HCI and complex failures (such as those caused by the incorrect reconfiguration of a system) are difficult to diagnose, especially when the collection processes are primarily focused on failure data such as crash dumps. As such, these processes (and the internal fault management system) had to evolve to improve the ability of engineers to diagnose the causes of all system failures.

Prior to the late 1990s, the traditional methods of col-

We can only fix
those bugs
we know about.

Automating Software Failure Reporting

lecting failure data were dependent upon the company having its own service arm and developing and maintaining a means of communicating with its users. Today the Internet provides a mechanism for software producers of any size to have an affordable method of communication with their users.

It seems obvious that all companies should develop a process to collect customer failure data and to distribute patches to fix any problems. Such a process benefits both the software producer and the end user. Unfortunately, a badly thought-out process can produce vast amounts of data that cannot be analyzed and at the same time can alienate the customer base. The following tale is an excellent example of how things can go wrong.

DEC (Digital Equipment Corporation) wanted a better understanding of why system managers were rebooting (for every single crash a system would have 10 reboots). During system reboot, the *why boot* process asked the system manager for the reason for the reboot. The response was captured in the system event log that DEC subsequently collected. The problems became evident when the process was rolled out to a few sites.

Most of DEC's customers set their servers to automatic reboot; although this is not perfect, it does often resolve issues (if the failure occurs as a result of the system running out of resources because of software leaking or aging, then a reboot will free up those resources—at least for a short time, thereby allowing use of the computer). Automatic reboots are particularly useful when the system manager is not available 24/7. The *why boot* process stopped the system from rebooting, as it required an input from the system manager. This resulted in long outages. Even when the system manager was present, *why boot* still caused problems. Installing a new application on a cluster might require reboots to a number of computers in sequence. During every reboot, the system manager would have to wait at the console to answer the

why boot question. Naturally, system managers were not inclined to answer the question with any great accuracy (usually the field was blank or filled with cryptic or not-very-polite comments).

The *why boot* program, although developed by an experienced computer manufacturer, is an example of a process that was expensive to develop and deploy, collected nonactionable data, and annoyed the customers. Thus the question: What issues need to be considered to avoid such disasters?

UNDERSTANDING YOUR USER BASE AND USAGE PROFILE

It is important to ensure that any data collected from the customer base is unbiased, or at least that any bias is understood. Some users are very good at filling in bug reports; they are usually technically competent and willing to go through the sometimes cumbersome and time-consuming process of filling in these reports. Though these users are invaluable, they are not necessarily representative of the user base (the more the product is targeted for home use, the less representative these users are). Therefore, to ensure an unbiased data set, the user interface of any process must be targeted toward the average user and should ensure that:

- The user interface is as simple as possible; a single button click is about the maximum acceptable for the average user.
- Requests for data collection must occur at a time that will not annoy users.
- The users will see each request for data; otherwise, they may become distrustful of the process, possibly viewing it as a form of spyware, and turn it off.
- The process must respect user privacy; otherwise, users will be reluctant to provide any information.

These objectives can be achieved only if the usage profile of your product is well understood. If the product is used in business and home environments, then your process may need to adapt to the different markets. Additionally, if the product can be used in a client or server environment, then the interface should change based on the context of the usage. For failures occurring in a client environment, the data request should be made to the current user; for a server, the request must be redirected to the system manager and the data collected will have to go through an authorized path.

DEFINING THE FAILURE PROFILE

A product fails when it does not conform to its specification. This formal classification of failures is not really

applicable in commercial products—users don't usually read the specification. The customer definition of a failure is that the product does not accomplish what the customer expected. While this failure classification may seem boundless, most customers do use common sense when defining failure. As such, crashes may not be the major cause of customer dissatisfaction. Users may be more frustrated by product behavior (such as requiring a reboot to correct an action), performance glitches, or confusing behavior.

Understanding the product package is also important in defining failure. From an engineering perspective, a product is bounded by its own software. Users may view things differently. For example, if a product fails to print, then the user may see this as a product defect even if the defect exists in a third-party print driver or operating system settings. This is especially true if all other products on the computer successfully print.

Typical indirect product failures are:

- User interface failures caused by illogical inputs or using the software in unintended ways.
- Using the software with components that are different from the recommended configurations.
- Hardware failures that corrupt storage.
- Software failures occurring in drivers or dependent third-party applications.

In addition, a product's failure profile is rarely static. New patches that fix known bugs and changes in the system configuration and hardware can all alter the environment. Analysis of Windows XP failures highlights the range of possible system configurations; currently there are more than 800,000 different kinds of plug-and-play devices on customer sites, with 1,500 devices being added every day. Additionally, there are 31,000 unique drivers with nine new ones being added daily (each driver has approximately 3.5 versions in the field, with 88 new driver versions being released daily). This is compounded by the average customer system continually changing—average speed is increasing at approximately five megahertz per week.

CAPTURING FAILURE INFORMATION

Predicting the information required for diagnosing failures is difficult. You should assume that the initial data set will be insufficient to diagnose all possible failures. Therefore, the process should be designed to evolve after its distribution to customers.

The following set of generic data helps diagnose most product failures. Unfortunately, in implementing this list, the engineer must also realize that there is a practical

limit to the amount of data that can be collected from the customer site (discussed later in this article).

Crash data captured in the product dump file and generated at the point of failure. As dump files can contain the total contents of the system memory, processing the dump file is often necessary to extract only the most relevant data.

System profile including the version of the product and the patches. Also useful are the versions of the hardware and other applications upon which the product depends.

Failure history. This is an important factor in helping diagnose product failures, specifically what happened to the system just prior to the failure. Many product failures are induced by external events (e.g., configuration changes, failures to other parts of the system, etc.).

User input. In general, you should avoid any manual input, as it may result in skewed or no data (users may get annoyed at such requests), but if, very occasionally, additional information is required, users are generally happy to provide it.

PRIVACY

Identifying the data that needs to be captured is not a purely technical problem; another important factor is privacy. Privacy laws vary greatly around the world, but you should assume that collecting personal data without user permission is illegal. Even if you ask for permission, legal issues still apply to the way personal data is stored and managed. Therefore, for general-purpose data collection, none should be traceable back to the end user. If there is a need to correlate failure data against user profiles, then a different data collection process must be developed and targeted at customers who understand and accept the process.

Although collecting personal data should be avoided, it is essential to differentiate between multiple failures

Automating Software Failure Reporting

occurring on multiple systems and multiple failures on a single system. This can be achieved by building a signature based on the computer configuration. While not perfect, since configuration changes alter the signature, this appears to be the best practical solution.

PROCESSING AND COLLECTING FAILURE DATA

Collecting failure data requires a process resident on the customer's computer that detects processes and transmits the data. This can be done in many ways and is dependent upon the product's customer base. For Windows this process is enabled by a dialog with the user as part of the installation process. Once a system administrator logs onto a system for the first time following a system reboot, the operating system automatically checks if the cause of the system outage was a system crash. If so, it processes the system dump file, generating a mini dump and an XML file containing versions of all drivers on the system. This data is subsequently compressed.

A prompt then appears on the screen requesting the user's permission to send this data to Microsoft. If the user agrees, the data is sent via HTTP POST. This method also allows the process to send a response back to the user by redirecting the http request to a Web page containing possible solutions (this is discussed in the next section).

Some corporations may restrict internal computers from sending data externally to the company. This complicates data collection and often requires a two-stage process: one process automatically routes the failure data to a central system(s) within the corporation; a second process sends this data off-site. In this latter scenario, a second type of report may be necessary. This report would define a list of patches recommended for installation on all corporate systems.

ANALYSIS ENGINE

The failure data collected from customer sites is fed into

a process that analyzes, stores, and—if possible—feeds back information to the end customer. The collection and analysis process must be completely automated. The collected data is processed, and the analysis engine should use a set of rules to diagnose the causes of the failures. The analysis engine is continually updated by service and development engineers assigned to debugging failures. By categorizing and storing the collected failures, it is possible to focus the engineering effort on the most frequently occurring bugs.

On most products a small percentage of defects results in the majority of failures. As such, the analysis initially focuses on these failures both in finding a resolution to the defect (usually a patch) and in identifying future failures of this type. If a crash is caused by a known defect, then the reporting system should inform the users of the availability of a patch. This feedback mechanism encourages users to submit failure information.

Immediately following its release, Windows XP failures were heavily skewed. A very small number of bugs were responsible for the majority of customer failures. The analysis engine identified these crashes based on the specific place that the system crashed and which drivers were loaded on the system. The initial focus was the generation of patches with the assumption that this would result in a significant decrease in the total number of Windows XP crashes. In reality, the rate of failures caused by these bugs continued to grow, forcing us to rethink our patch distribution mechanism (this will be discussed in the next section).

Windows engineers then started to encounter crash categories that were not as easy to solve, and over time they developed several strategies to help debug these failures, specifically:

Improving the quality of the data collected from customer sites. For example, Windows XP SP2 will collect additional information with a focus on hardware (e.g., BIOS version, ECC status, and processor clocking speeds to identify overclocking). As Microsoft shares failure data with partners, a number of these companies now store manufacturing information on the system that is collected as part of the dump process (e.g., some manufacturers store the make and date of installation of every product).

Special tests to identify hardware failures. Tests have been written to identify hardware-related failures. For example, as part of the crash dump, several memory pages are captured that contain operating system code. These pages are verified to see if they have become corrupted. If corrupted, it is possible to identify the possible

causes and recommend solutions to the customer (e.g., if the corruption is hardware-related, the customer is pointed to a hardware memory checker).

Developing data-mining tools to assist in failure diagnosis. Engineers are assigned to a group of crashes that the analysis engine believes have a single cause. In addition to the data in the crash dumps, tools are available for the engineer to mine the crash database for other relevant information (e.g., identifying the frequency of the combination of specific drivers in other crash groupings).

As the engineers resolve the causes of crashes, the analysis rules are updated to identify all future crashes of this type. The percentage of Windows XP crashes that can be automatically resolved through the analysis engine continually fluctuates. While patches are released to resolve current issues, new drivers and peripheral devices of various qualities continually appear. Determining the ideal diagnosis rate is difficult; diagnosing a high percentage of bugs may simply indicate that the patch distribution process is broken.

Of the currently diagnosed Windows XP failures, 5 percent are Microsoft software bugs, 12 percent are the result of hardware failures, and 83 percent are third-party failures.

The pie charts in figure 1 provide a breakdown of the causes of hardware and driver crashes.

The percentage of hardware failures appears to be increasing, probably as a result of the aging profile of the systems running Windows XP (new systems bought when XP was released are now three years old). As previously mentioned, the failure ratios of third-party drivers vary over time, because the release of a new version of a popular driver may result in an increase or decrease in failure rates.

Sometimes the solutions to these problems can be embarrassing. For example, analysis of many driver crashes showed that the root cause was that the drivers did not check for error conditions following

a system call. In discussions with the developers, they said that code had been copied from help files in Microsoft's Drivers Development Kit and MSDN online documentation. The original documentation was written to provide succinct examples of how to use the system calls and so lacked error handling. The fix: the documentation has been rewritten to include checks for error conditions.

PATCH DISTRIBUTION

The Microsoft analysis engine is designed to correct common crashes by providing a patch facility when the crash signature matches. An important issue is whether the patch should be distributed to all customers through a generic patch distribution process, thereby preventing other system crashes. The development of an effective patch distribution process is complex. Here are a few factors to be addressed in developing a successful process:

Patch quality. Even though the inclination is to release the patch as soon as possible, especially if the patch addresses security defects, it should be thoroughly tested in as many user environments as possible, prior to distribution. If a patch subsequently causes problems, then customers will be less inclined to install future patches.

Versioning. This is necessary to verify which patches have been applied to the system. Versioning is important where patches have dependencies on other patches, as the version information is required to resolve these dependencies.

Automating Software Failure Reporting

Patch size. The smaller the size of the patch, the smaller the load applied to the server to download the patch—and the less time it will take the end user to download and install the patch. A good test is whether the patch can be practically downloaded by users with 28k modems.

Number of patches per year. Releasing as many patches as possible for all known failures may seem logical, but this can place a great load on end users. Corporations have to test and stage each patch prior to its release. The process should release patches only for problems that affect a noticeable percentage of users.

Installation. A patch should be installed through a single button click and have as minimal an impact on the end user as possible.

Automatic deployment for the home user. For Windows, Microsoft distributes patches through an automatic pull process (the process checks the Microsoft site to identify if critical patches are available and, if so, downloads them as a background process). This was deemed necessary to deploy security patches to as many users as possible. For deployment of bugs to noncritical applications, a less invasive patch distribution is probably preferable.

Method of distribution. Patches can be distributed through the analysis engine or other processes. If users run a Windows update, then they will see the set of patches that are critical, as well as those that are recommended, based on the system configuration.

Staged deployment for corporate users. For ease of management, some businesses want their employees to run a common set of software at a common version. To control the environment, corporations may prefer to distribute the patches themselves. As part of this process they will stage and test the patches within their own environments, prior to distribution.

The patch distribution process may be the most important part of the whole process, since there is little

point in generating patches if no customers subsequently install them. One possible method of ensuring patches get installed at customer sites is to encourage computer manufacturers to pre-install all patches as part of their build processes.

BUG REPORTING SYSTEMS ALWAYS CHANGING

A successful bug reporting system is hugely beneficial to both software producers and users. It can provide an excellent understanding of the quality of a company's products, while giving customers an improved user experience. Developing such a process is complex, however, and if done incorrectly can do more harm than good in terms of customer satisfaction.

Developing a failure reporting system requires an understanding of a product's customer base, as well as usage profile. It is also important that you understand the failure profile of your product so that you can focus on events most annoying to your customers. Although product attributes are unique, there is a generic set of data that, if collected, will help in diagnosing failures. In collecting customer data, however, you must address all privacy concerns prior to rolling out your process. Along with a failure collection system, you also need a process that can distribute patches to address those failures.

At Microsoft, our experience has led us to develop a generic methodology to process, transmit, analyze, and respond to customer failure data. Differences exist in the way this process can be implemented; it is usually dependent upon the product type and its failure profile.

While the development of this system has been hugely beneficial to our organization, the usage profiles of computers are continually changing, as their configurations evolve. Thus, their failure profiles will never be static. As such, no matter how successful an automatic software failure reporting process is, it will always need further development. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

BRENDAN MURPHY is a researcher at Microsoft Research Center in Cambridge, UK, where he specializes in system dependability, including failure prediction, system fault management architectures, application availability, and cluster reliability and availability. Prior to joining Microsoft, Murphy worked at Compaq and Digital. He is a graduate of Newcastle University.

© 2004 ACM 1542-7730/04/1100 \$5.00