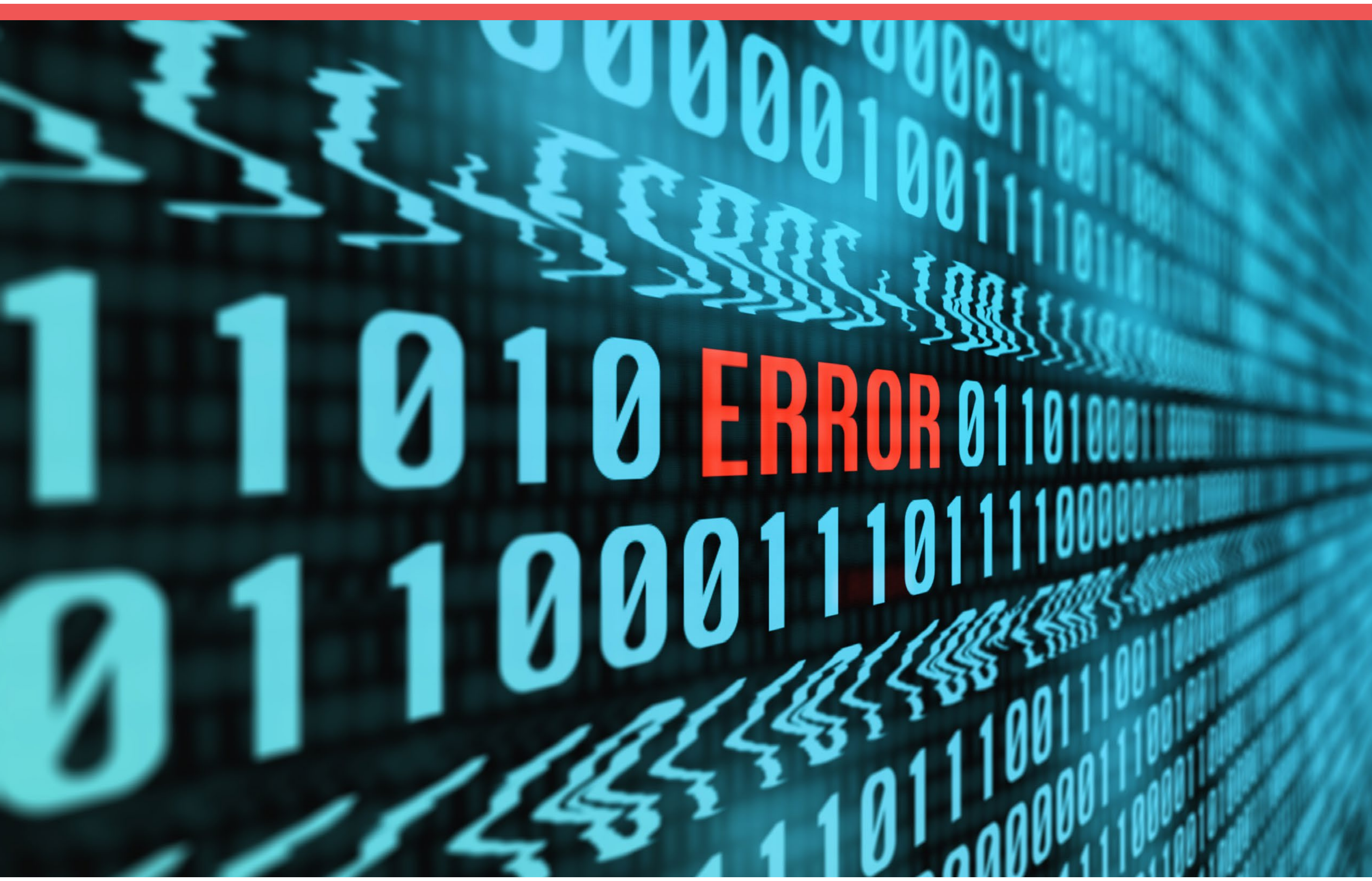
WHITEPAPER

Visual Studio MAGAZINE | ADTMAG APPLICATION DEVELOPMENT TRENDS



ACCELERATING ERROR DETECTION AND RESOLUTION FOR DEVELOPERS

By Peter Varhol

Backtrace

www.backtrace.io

Finding the root cause of application exceptions is one of the most difficult and frustrating jobs developers face. It's challenging to find time to resolve issues during the development process, when delays cost the team and the company time and money; but the longer a team waits to address bugs, the more difficult they are to resolve.

While testing has become more sophisticated, it covers only a fraction of the functionality, and pre-production test environments cannot replicate the complexity of a distributed, cloud-based production system. And outside of the cloud, end users continue to use traditional native applications, mobile systems, TV apps, and new embedded devices, with hundreds of different brands and operating system versions. This leads to more complexity, with the sheer number of heterogeneous runtime environments that end users will use.

When testing fails to identify issues that occur in real-world usage, it leads to application crashes and production downtime that require an immediate response. In many cases, any amount of downtime will cost the business money, customers, or goodwill. In an all-hands on deck situation, developers need powerful solutions to support them in quickly diagnosing and repairing application errors.

WHAT CAN GO WRONG?

There are a wide variety of application defects and exceptions that can occur in both unmanaged and managed code. Understanding the types and causes of exceptions possible is the first step in building a strategy to deal with them.

Most faults leave behind clues as to their cause. For unmanaged—or low-level code—like C and C++, those clues may come in the form of dump files and similar artifacts that include raw data and contents of memory locations at the time of the crash. Searching these dumps for clues to the cause of a crash is similar to looking for a needle in a haystack. The amount of data is overwhelming, and requires the time and unique expertise that many developers don't possess.

For these low-level languages, buffer overflows and pointer mismatches are common. It's not unusual for a developer to make a mistake on pointer arithmetic or variable size, mistakes that can cause an overwrite of data or code. Some other common types errors with the potential to crash a

“REDUCING THE TIME NEEDED TO DIAGNOSE AND ADDRESS EXCEPTIONS IS VITAL IN PRODUCTION.”

C/C++ based application include:

- A malformed conditional expression, when it’s inadvertently not tested for or handled in the conditional construct, can lead to a fault.
- Memory leaks, when a memory location isn’t reclaimed for future use once it has served its purpose, can cause instability and faults. While most applications call on other memory addresses for storage, these leaks accumulate after an application has run long enough and the application will run out of heap space and crash.
- Segmentation faults, where an application tries to access a protected area of memory, are often the result of poor pointer arithmetic and virtual memory addressing, and can be particularly difficult crashes to analyze and address.
- Another common type of fatal pointer error is the null pointer, where the application expects a valid value at that pointer address, only to discover that the pointer hasn’t yet been set.

Many developers have come to rely on managed platforms such as the .NET Common Language Interface (CLI) to reduce the likelihood of the aforementioned fatal exceptions. The CLI automates a number of error-prone operations, and will catch and perform basic processing on several common exceptions. Other higher languages such as JavaScript, Swift, or Go have similar runtime environments that provide functions like exception handling, memory allocation, and garbage collection.

“UNDERSTANDING THE TYPES AND CAUSES OF EXCEPTIONS POSSIBLE IS THE FIRST STEP IN BUILDING A STRATEGY TO DEAL WITH THEM.”

However, as many developers using managed platforms and languages can attest, exceptions in logic are still possible. Fatal arithmetic errors, such as divide by zero, are common when appropriate checks are not put in place, and are common in both managed and unmanaged languages. Anything that causes an application to produce an illegal or nonsensical result has the ability to cause an exception, depending on the context and the code following. It’s not feasible to catch every possible exception, and some will leak through to crash or otherwise impair the application.

Some developers meticulously add appropriate try-catch-throw statements to try to recover and continue execution, or the application might simply provide an error message that describes the exception and exit as gracefully as possible. What developers more often see when a system or application crashes is a cryptic and usually unhelpful error message. Encapsulating all code into a try-catch-flow block is inefficient in both developer time and application execution, and simply isn’t practical for large-scale use.

Developers can also employ log analysis tools to attempt to make sense of crashes. Servers log a great deal of data on both the system and the application, and developers can analyze that data to help diagnose crashes. However, developers can’t easily correlate log data with application activities and state, making it difficult to find root causes with existing solutions.

In many cases, developers and operations professionals may not even be able to determine the cause of an exception, even if it is caught. If an exception is sporadic and seemingly random, it may be impossible to reliably reproduce the error, and any reproduction may seemingly come under different situations. Furthermore, in multithreading systems, race conditions can cause sporadic and random exceptions. Situations such as this, as well as the inevitable need to get back online in production, may lead developers to simply reboot the instance rather than investigate the exception. That means that the root cause is never addressed, and exceptions will continue to affect the application.

“DEVELOPERS CAN’T EASILY CORRELATE LOG DATA WITH APPLICATION ACTIVITIES AND STATE, MAKING IT DIFFICULT TO FIND ROOT CAUSES WITH EXISTING SOLUTIONS.”

GET BACK UP AND RUNNING AS QUICKLY AS POSSIBLE

Teams should strive to reduce the time needed to diagnose and resolve exceptions. While this is important during development, it is vital in production. Holding up efforts during development can have schedule and cost implications, but preventing the application from operating in production can be catastrophic to a business. This is also the case for systemic issues—those that appear multiple times in similar form. Recognizing the pattern can be difficult for developers unless presented in specific ways.

Developers must have an understanding of the operation of the application and use of the programming language, as well as what kinds of exception data is produced by the language and platform. Reducing the time for diagnosis and repair involves following a structured process for recovering and analyzing exception data.

The first thing developers can do during the development process is to produce a debug build with symbols preloaded. While not always appropriate for production deployment, it can be a significant help during development and test, especially in a CI/CD environment, where much of the workflow is automated and continuous.

For production environments, external symbol files provide additional information to the exception, making it human readable and linking back to source code if available. They organize the call stack information in a logical sequence to make it more understandable. When debug builds are not available, appropriate use of symbol files helps make the problem solvable. Loading and using symbols should be made as automatic as possible in the workflow in development, build, and test to reduce the time to detection and resolution.

INTEGRATING EXCEPTION ANALYSIS WITH THE LARGER WORKFLOW

Development and DevOps teams incorporate many tools to improve their workflow, including ones focused on communication, monitoring, and ticket tracking. Integrating exception analysis with these existing tools is critical towards reducing mean time to detection and resolution.

Exception analysis data can be used to automatically trigger notifications and alerts, as well as create tickets.

- **Communication:** Coordination between team members is an important aspect of exception analysis. The expertise needed to analyze data and produce an answer may reside in individuals and locations beyond the immediate crash, and a seamless way of communicating data using tools such as Slack or Webex Teams helps automate and accelerate diagnosis and repair.
- **Monitoring:** If the application is in production, in all likelihood the enterprise is monitoring it for performance and availability. The monitoring might be as simple as scanning the log files, or it might be a comprehensive view of all transactions, timings, and exceptions, fatal and non-fatal. These include data that would likely be useful in evaluating unhandled and fatal exceptions. A business-critical production application also has to alert teams, especially DevOps teams, that an exception or crash has occurred. Integration with common monitoring tools should also be a high priority for exception and crash analysis.
- **Tracking:** While exceptions and crashes have to be analyzed and addressed quickly, especially when they occur in production, the details of such an event still need to be recorded and tracked for further analysis and historical purposes. Therefore, integration with bug tracking or planning tools such as JIRA or GitHub is a strong requirement so that teams can keep a lasting and detailed record of the crash, actions taken, and results.

“DEVELOPERS NEED SOLUTIONS TO MANAGE CRASH AND EXCEPTION DATA TO BETTER UNDERSTAND WHERE THEY OCCUR AND PREVENT FUTURE OCCURRENCES.”

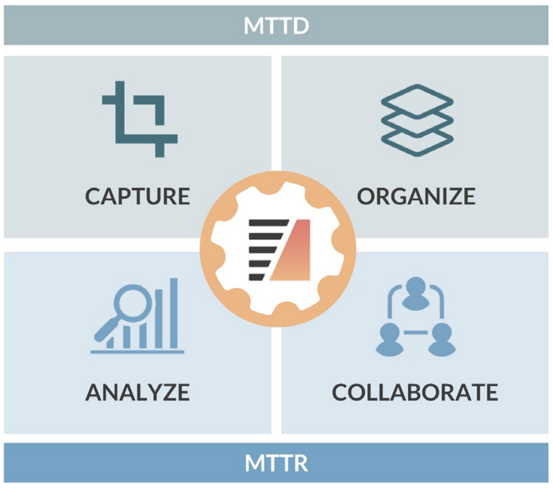
IMPROVE YOUR EXCEPTION HANDLING

Developers need solutions to manage crash and exception data to better understand where they occur and prevent future occurrences. This is particularly important as applications reach larger scale, where the pressure to get from crash back to production is exponentially greater, and it’s imperative that people collaborate efficiently, even across teams.

Solutions and techniques that make crash and exception data more transparent are essential in accelerating analysis and resolution. This is because exception data is exceedingly difficult to understand and use in diagnosis. It typically involves cryptic messages, code and data in hexadecimal format, along with errors from the operating system and platform.

Integrating communication tools and supporting the collaboration among team members is essential to the analysis and resolution efforts. Sharing data within the tool chain, and with other developers who may have expertise and ideas, serves to get answers more quickly, and to implement and try out those answers.

A structured examination of crash and exception data can accelerate the resolution of errors and get applications back online quickly. The exact nature of the structure depends on the application, programming language and platform, and what kinds of errors it produces. Often



crash data can be used in conjunction with other monitoring and diagnosis information to minimize downtime and keep an application deployed.

Backtrace provides the ability to organize, query, and readily obtain valuable information on the state of the system at the crash point. Using Backtrace, crash and exception reporting can be done across the majority of desktop, server, mobile, and embedded Operating Systems. This includes Windows, MacOS, iOS, various flavors of Linux (including Android and Yocto), and FreeBSD. Backtrace will consume a crash report object and crash dump file, such as minidump or core dump, that is generated by an application fault.

Backtrace can consume the output from common minidump generators, like Crashpad, Breakpad, or Windows, that produce a snapshot of the application's state at the time of crash, including threads and their call stacks, the stack memory space for each thread, register values, attributes (environment variables, system information, and custom metadata), and a listing of loaded modules with their metadata. Once consumed, Backtrace analyzes the state of the application at the time of error to support a quick resolution.

Backtrace captured libraries do not run until the moment an error occurs. This means there is no performance impact during normal application execution, making Backtrace perfectly suited for production environments. When implemented during testing, Backtrace allows teams to fully investigate errors without the need to recreate the issue. In production environments, Backtrace streamlines the process of capturing and processing relevant exception and crash data, giving you visibility into the issues affecting your end users.

Visit http://backtrace.io for more information on its features and capabilities, and to schedule a live demonstration.

Find out more: http://backtrace.io

Backtrace

Peter Varhol is a technical evangelist. He has more than 20 years of experience writing technical articles, blog posts, and papers on technical practices and products. He has graduate degrees in computer science, mathematics, and psychology, and experience as an evangelist, product manager, software developer and tester, and technology journalist.