Active Server Availability Feedback

James Hamilton

Microsoft SQL Server
One Microsoft Way
Redmond, WA
USA
JamesRH@microsoft.com

Abstract

The current software development process in common use within industry is inefficient, in that the time required to incorporate results from competitive, beta, and previous releases into new versions available to customers is typically measured in years. Further, the accuracy of customer feedback returned to the development team is frequently weak or incomplete, with samples often drawn from only a small, self-selected set of customers. This paper argues that we can automate this feedback process and, in so doing, drive an order of magnitude improvement in the rate at which software evolves and improves.

1. Introduction

The author has worked for a decade and a half on commercial system software, first language compilers and later database management systems, at two of the three leading commercial database system producers. Over this period, we have experienced a ten fold increase in the size of these products when measured in lines of code, and have seen an expansion both in development and test team size that roughly parallels code base growth. Contemporary database products are typically larger than three million lines of code and the engineering teams for mature, industry leaders have grown to several hundred engineers each. For this system size and complexity growth to even be possible, it is very clear that there have been incremental improvements to the software development process, which we fully expect will continue. What is less clear is: 1) do systems really need to be this big to meet current customer requirements, and 2) could these systems be evolved more quickly to respond to customer requirements in a more targeted fashion?

Systems growth and development process improvement will continue, but existing processes only allow our current understanding of customer requirements to be translated into software. The improvements do nothing to increase the quality of our understanding of customer requirements nor do they help to tighten the feedback loop between a customer's experience using the product and a subsequent improvement to that product.

Proceedings of the 2003 CIDR Conference

2. Defining an Efficient Software Development Process

Clearly, as an industry, we should continue working to improve the software development process – there is no question as to the worthiness of this endeavour, but there is considerable debate on whether the bulk of the engineering work that these process improvements are enabling are actually accurately targeting customer requirements. Is the feedback between customer experience and subsequent product improvement sufficiently responsive? In this paper we focus on improving the feedback loop between a customer's actual experience using a server-side software product and the release of changes to the product that subsequently improve this experience.

In the mid-1960's a group of researchers, including Nobel laureate Paul Samuelson, defined the Efficient Market Hypothesis [10]. This theory argues that market prices reflect the knowledge and expectations of all investors. An efficient market is one that translates knowledge very rapidly into an adjusted and accurate stock price. There have been several instances of late where critical market information was not made available to investors and, as a consequence, market prices did not accurately reflect the true value of the companies in question. However, the theory is still of value, in that any market that rapidly translates deep knowledge of a given company's present and future value into an accurate market price is an efficient market.

Let us define an efficient software development process similarly, as one where customer needs and wants, including their (possibly negative) experiences using existing products, are translated into targeted software improvements that are quickly made available to customers. An efficient software development market is one where requirements and issues with existing products are rapidly responded to with targeted product improvements.

Our work here focuses on improving the bandwidth and speed of the communication between customers and the product development team, but does not address new requirements gathering or better understanding of where additional features would help customers. Focusing on this feedback loop from customers to the product team and the translation of this feedback into targeted product improvements, we note that the current processes in widespread use today are extremely weak, which is to say that the opportunity for improvement is great. Current customer feedback to product teams suffers from low bandwidth channels, and few customers actually participate. Looking at the sources of customer feedback available to a development team, we have the marketing team reporting back on why they were, or were not, able to close a sale, the product support team reporting on common customer problems, user group presentations made by customers, consultant and industry analysis reports, and beta customer feedback. In addition, most development teams closely partner with a small subset of their customers with whom there is a high quality exchange of information. Summarizing what is available, a few problems can be immediately seen: 1) most data is often not directly obtained from customers in that it's reported through (a possibly biased) intermediary with a significant time lag, and 2) the data which is directly sourced from customers is very expensive to gather and, as a consequence, is only obtained from a small fraction of the customer base with enterprise customers being significantly over-represented.

We propose using software data collection systems to solve these problems. Automatic data collection can allow direct, detailed customer feedback to be obtained economically from a substantial cross-section of the entire customer base which enables reliable statistical analysis. This gets reliable and detailed feedback directly back to the development team, allows rapid product feature or improvement prioritization to be made, is a far more precise process, and the information can be acted on much more quickly.

Our work on active server feedback is being driven by two basic premises: 1) software availability will only be incrementally improved by continued investment exclusively in existing approaches to improving software quality, and 2) system downtime and the causes of these losses of availability are not sufficiently well-understood and, as a consequence, are not fully and efficiently addressed since it is not possible to effectively address a problem when the root cause and problem magnitude are not fully known.

Why do we feel that current techniques can only yield incremental improvements in software quality? The first and perhaps strongest argument is one based upon the sheer size and complexity of a modern server-side software stack. A modern network-attached storage sub-system is currently over a million lines of code. Operating systems, logically above the storage sub-system, are now tens of millions of lines of code with systems such as Windows XP reportedly approaching fifty million [3]. Continuing to examine the server-side software stack, a typical data management system is over three million lines of code. Additionally, a high scale, mission-critical application like SAP is over thirty-seven million lines of code [9]. What this means is that for a customer to experience a reliable, robust system that returns correct results in a predictable manner, this entire hundred million line software stack must all operate correctly. What makes this even more difficult to achieve is that all components in the stack are on different release cycles, there is no integration test team responsible for the entire stack, the stack is usually sourced from multiple vendors and all vendors are asynchronously releasing possibly non-cooperating fixes. As an industry, we attempt to deal with this complexity by depending upon well-architected interfaces between components and by investing heavily in testing efforts. On systems software teams on which we have worked over the years, tester to developer ratios have approached 1:1, yet it is clear that the core complexity problem is neither solved nor is the impact substantially mitigated. Further investment in maintaining and improving the software quality assurance process is clearly money well spent, and incremental improvements will continue to be realized, but a fundamental improvement in product availability will require new processes and approaches.

One approach that appears to have merit is capturing actual operational data from the field and feeding this customer experience back into the development process. An application of this approach that has yielded good results is to gather customer problems as reported to the vendor service organization [1, 2, 11]. A similar approach is to analyze problems as recorded directly by the customer [7, 8]. The principle advantage of these tracking systems is that, when an error is reported to a vendor or explicitly tracked by a customer, it is normally a serious event and therefore interesting. However, our data suggests that administrative error is a leading cause of operational system downtime and this entire class of errors are typically not reported to system vendors as a bug and are rarely accurately tracked in customer reports, since the accuracy and completeness of these reports are wholly dependent upon the administrators themselves, many of which do not do a complete job of self-evaluation.

A refinement to this error tracking approach that depends upon customer reports is one where system-generated error logs are analyzed [4, 5, 6, 13]. These failure-tracking systems have access to the full range of failures from hardware and software issues through to administrative errors but the information that can be mined from an error log is only a small subset of the information about the state of a failing system. As a consequence, it can be very difficult to ascertain the real causes of many failure modes. Error log analyzers have a more complete view of system availability, but a less precise view over the causes of the system failures, than do those systems dependent upon actual customer-initiated feedback. Some researchers have combined event log analyzing with administrator interviews to improve the precision of the error classification [6]. This can provide quite precise downtime cause classification, but these research methods are people-intensive in that interviews must be conducted and the techniques tend not to scale cost-effectively to very large customer sets. In addition, many of the antecedent conditions to failure are not errors so they will not appear in logs. As a result, these systems have limited power in finding correlations between operationally-acceptable system states and subsequent system failures. Predictive failure analysis with this subset of the system state data is necessarily incomplete. In this paper we focus on two forms of improvement: 1) reducing human involvement and subsequent cost in the data gathering process, and 2) improving the detail of the information gathered to include administrative errors and non-failure state tracking to help support predictive failure models.

We refer to failure and system state tracking systems that return data back to the development team without human intervention, interviews, or site visits as Active Server Availability Feedback (ASAF) systems. In this paper and we will look at two ASAF systems in more detail. The first system, Watson [12], is already in broad use in client-side products and is now being adapted for server-side deployment. In Section Three of the paper, we will describe Watson and show how it has been adapted for use with Microsoft SQL Server 2000 Service Pack Three. In Section Four, we describe the Data Collection Agent (DCA), which is a research system currently deployed on over 100 Microsoft SQL Server 2000 production servers. The Data Collection Agent is an ASAF system that provides much more detailed data than Watson on product usage and the availability achieved and, rather than only reporting failures, it allows many system metrics to be monitored continuously over time. We believe this additional detail will allow correlation and trend analysis, allowing us to learn more about the causes of downtime and how to efficiently address it. In Section 4.1, we outline some early results from the DCA project.

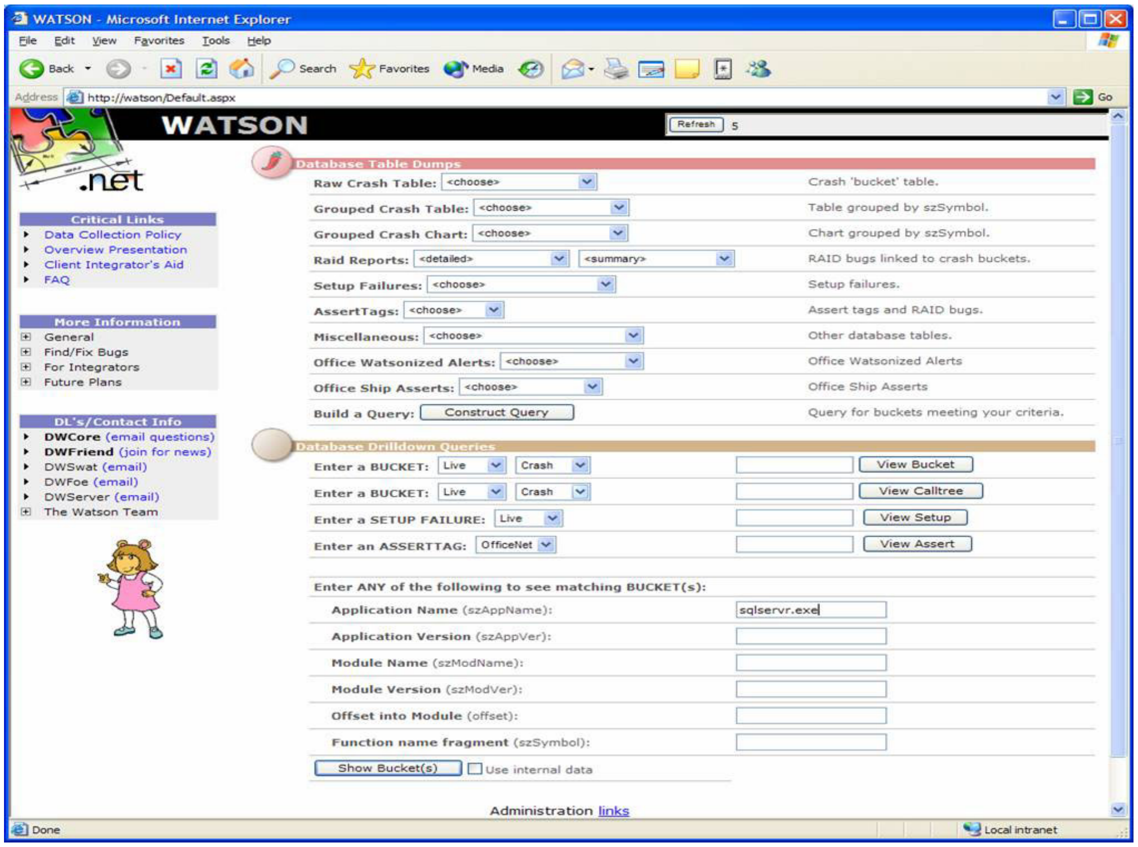
3. Watson: System Failure Reporting

Watson is an error reporting framework originally developed by the Microsoft Office team but now in use by Windows XP, Internet Explorer, MSN Explorer, Visual Studio 7, and other products in addition to the Office suite. It is a multi-tier system that automatically returns to the development team reports on failures experienced by customers along with sufficient information to diagnosis many of these failures.

Each software system under Watson monitoring has an additional software component responsible for detecting and reporting failures back to the data collection system. The backend system is composed of many cloned IIS web servers, each of which stores data across a firewall into a SQL Server database and a file store for the bulk debug information. The workload is distributed over the IIS systems by Windows Load Balancing Server. The backend database server is replicated for redundancy and the data is post processed into a system that is accessible to the product development team.

When a failure is detected, the user is informed and given the opportunity to send the failure data back to the development team. For privacy protection reasons, we first prompt users on whether they would like to send the failure data and this dialog defaults to “do not send” to reduce the likelihood of accidental transmission. There was considerable concern early on in the development of Watson that customers would be unwilling to return failure data. However, we have learned in use that a substantial percentage of customers are very motivated to see the products they use improve and, as a consequence, are willing to partner with the development team and return this data. Encouraging customer participation has not been difficult and, although we do not have statistical analysis to support the assertion, we believe that we do receive feedback from a sizable and statistically valid cross-section of the customer base.

Since we have the same privacy concerns on the server side as we do on the client, it is very important that participation in the program be optional and the default be not to participate. However, unlike the client side, it is not practical to ask the customer if they are willing to send failure data using a pop-up window on a server-side system. Addressing this concern, we prompt for participation at server install or upgrade time and persist this setting until subsequently changed. We have not yet had enough experience with this approach to know whether this particular form of “opting in” reduces participation. We expect that it may but, even if participation dropped to 10%, were that a valid cross section of the customer base, it would be more than adequate for these purposes.


WATSON - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://watson/Default.aspx
Go
WATSON
Refresh
.net
Done
Local intranet

5

Critical Links
- Data Collection Policy
- Overview Presentation
- Client Integrator's Aid
- FAQ

More Information
- General
- Find/Fix Bugs
- For Integrators
- Future Plans

DL's/Contact Info
- **DWCore** (email questions)
- **DWFriend** (join for news)
- DWSwat (email)
- DWFoe (email)
- DWServer (email)
- The Watson Team

Database Table Dumps

Raw Crash Table:	<choose>	Crash 'bucket' table.
Grouped Crash Table:	<choose>	Table grouped by szSymbol.
Grouped Crash Chart:	<choose>	Chart grouped by szSymbol.
Raid Reports:	<detailed> <summary>	RAID bugs linked to crash buckets.
Setup Failures:	<choose>	Setup failures.
AssertTags:	<choose>	Assert tags and RAID bugs.
Miscellaneous:	<choose>	Other database tables.
Office Watsonized Alerts:	<choose>	Office Watsonized Alerts
Office Ship Asserts:	<choose>	Office Ship Asserts
Build a Query:	Construct Query	Query for buckets meeting your criteria.

Database Drilldown Queries

Enter a BUCKET:	Live	Crash	View Bucket
Enter a BUCKET:	Live	Crash	View Calltree
Enter a SETUP FAILURE:	Live		View Setup
Enter an ASSERTTAG:	OfficeNet		View Assert

Enter ANY of the following to see matching BUCKET(s):

Application Name (szAppName):	sqlservr.exe
Application Version (szAppVer):	
Module Name (szModName):	
Module Version (szModVer):	
Offset into Module (offset):	
Function name fragment (szSymbol):	

Show Bucket(s) ☐ Use internal data

Administration links

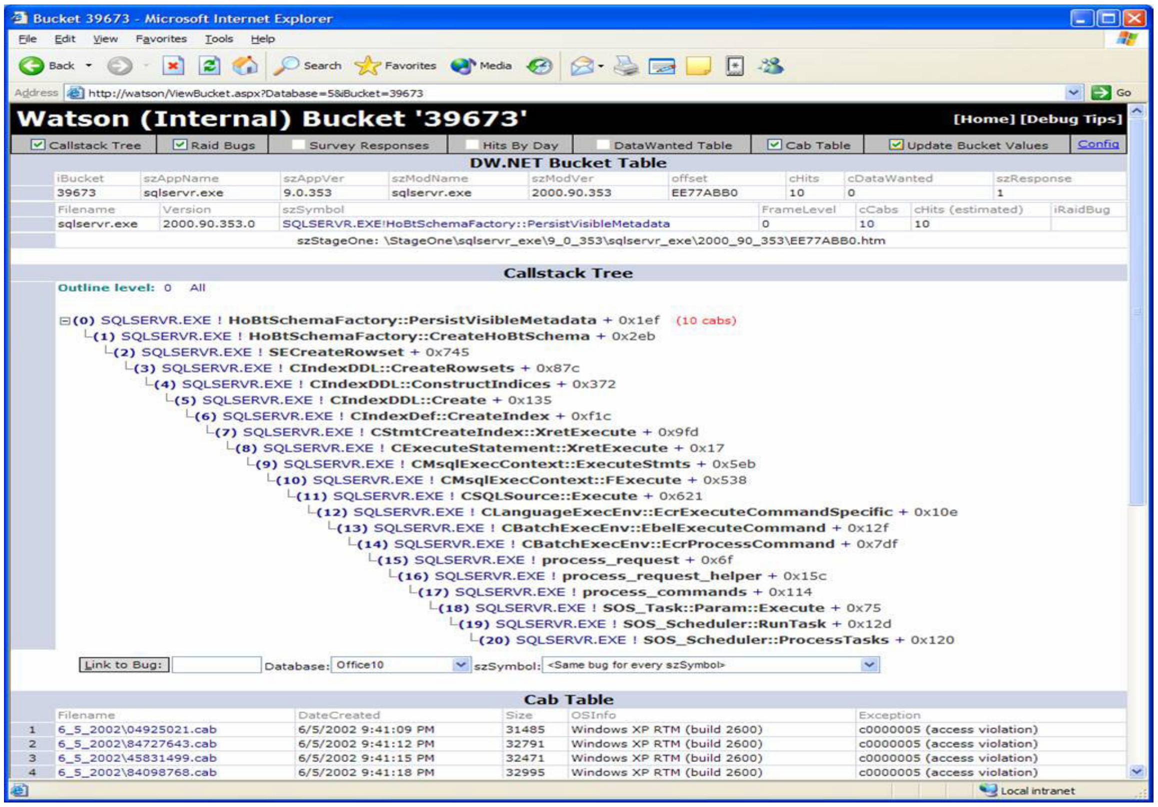
Figure 1: Internal Watson Query Page

In SQL Server, we have Watson enabled for product installation failures, errors found by the core engine, the data replication sub-system, the OLAP engine, and the system management tools. On setup failures, we return the log from the setup execution. On operational-server error conditions, we return the current point of execution and the call stack that got there, system configuration information, the modules currently loaded into the server address space, the type of exception if applicable, and global and local variable state.

A key to managing potentially large amounts of data is to have an efficient and accurate means of aggregating the results into failure classes. Rather than looking at all failures individually, we want them sorted into unique failures and we want the count of all instances of that failure. In essence, we need a signature or unique name for the failure. In SQL Server, we use a hash of the stack trace on the basis that all failures at a given point in the code, with identical call stacks to reach that point, are very likely to represent the same error. So, for every unique stack trace, we have a count of the number of instances of that issue that have been reported. This allows us to concentrate first on those issues with the greatest number of instances and, as one might guess, the distribution is very highly skewed, with a small number of issues causing the vast majority of the reports. This is the ideal situation where a moderate engineering investment has substantial leverage. When development investment is driven by better quality information, we are able to address issues much more quickly and much more efficiently.

In addition to using this bucketized design, where we use a stack trace hash as a means of counting the number of instances of a unique error type, we can also use this problem identification mechanism to return custom information back to users. Rather than always returning the default "thanks for contributing to the improvement of the product", for those issues where we have previously identified that a code change is required to fix the problem, we can return an explanation with a URL to the appropriate QFE (Quick Fix Engineering). In these cases, we are both tracking the failures and helping customers get the fixes they need more quickly.

Continuing to build on the bucketized design, we also leverage this tracking system as a means of restricting the amount of data that is sent. If we already have several instances of a given failure and believe we do not need further stack data to determine the cause, we can configure the system to count future instances of the problem without sending all the system state and debug


Bucket 39673 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://watson/ViewBucket.aspx?Database=5&iBucket=39673
Go
[Home] [Debug Tips]
✓ Callstack Tree ✓ Raid Bugs Survey Responses Hits By Day DataWanted Table ✓ Cab Table ✓ Update Bucket Values Config
Local intranet

Watson (Internal) Bucket '39673'

DW.NET Bucket Table

iBucket	szAppName	szAppVer	szModName	szModVer	offset	cHits	cDataWanted	szResponse
39673	sqlservr.exe	9.0.353	sqlservr.exe	2000.90.353	EE77ABB0	10	0	1

Filename	Version	szSymbol	FrameLevel	cCabs	cHits (estimated)	iRaidBug
sqlservr.exe	2000.90.353.0	SQLSERVR.EXE!HoBtSchemaFactory::PersistVisibleMetadata	0	10	10	

szStageOne: \StageOne\sqlservr_exe\9_0_353\sqlservr_exe\2000_90_353\EE77ABB0.htm

Callstack Tree

Outline level: 0 All

```
⊟(0) SQLSERVR.EXE ! HoBtSchemaFactory::PersistVisibleMetadata + 0x1ef   (10 cabs)
  └(1) SQLSERVR.EXE ! HoBtSchemaFactory::CreateHoBtSchema + 0x2eb
    └(2) SQLSERVR.EXE ! SECreateRowset + 0x745
      └(3) SQLSERVR.EXE ! CIndexDDL::CreateRowsets + 0x87c
        └(4) SQLSERVR.EXE ! CIndexDDL::ConstructIndices + 0x372
          └(5) SQLSERVR.EXE ! CIndexDDL::Create + 0x135
            └(6) SQLSERVR.EXE ! CIndexDef::CreateIndex + 0xf1c
              └(7) SQLSERVR.EXE ! CStmtCreateIndex::XretExecute + 0x9fd
                └(8) SQLSERVR.EXE ! CExecuteStatement::XretExecute + 0x17
                  └(9) SQLSERVR.EXE ! CMsqlExecContext::ExecuteStmts + 0x5eb
                    └(10) SQLSERVR.EXE ! CMsqlExecContext::FExecute + 0x538
                      └(11) SQLSERVR.EXE ! CSQLSource::Execute + 0x621
                        └(12) SQLSERVR.EXE ! CLanguageExecEnv::EcrExecuteCommandSpecific + 0x10e
                          └(13) SQLSERVR.EXE ! CBatchExecEnv::EbelExecuteCommand + 0x12f
                            └(14) SQLSERVR.EXE ! CBatchExecEnv::EcrProcessCommand + 0x7df
                              └(15) SQLSERVR.EXE ! process_request + 0x6f
                                └(16) SQLSERVR.EXE ! process_request_helper + 0x15c
                                  └(17) SQLSERVR.EXE ! process_commands + 0x114
                                    └(18) SQLSERVR.EXE ! SOS_Task::Param::Execute + 0x75
                                      └(19) SQLSERVR.EXE ! SOS_Scheduler::RunTask + 0x12d
                                        └(20) SQLSERVR.EXE ! SOS_Scheduler::ProcessTasks + 0x120
```

Link to Bug: Database: Office10 szSymbol: <Same bug for every szSymbol>

Cab Table

	Filename	DateCreated	Size	OSInfo	Exception
1	6_5_2002\04925021.cab	6/5/2002 9:41:09 PM	31485	Windows XP RTM (build 2600)	c0000005 (access violation)
2	6_5_2002\84727643.cab	6/5/2002 9:41:12 PM	32791	Windows XP RTM (build 2600)	c0000005 (access violation)
3	6_5_2002\45831499.cab	6/5/2002 9:41:15 PM	32471	Windows XP RTM (build 2600)	c0000005 (access violation)
4	6_5_2002\84098768.cab	6/5/2002 9:41:18 PM	32995	Windows XP RTM (build 2600)	c0000005 (access violation)

Figure 2: Watson Issue Detail Display

information that we normally transmit. In addition to being able to request less information on a given error condition, we can also chose to request that more be sent. If a particularly difficult problem is being debugged, the server-side Watson systems can be configured to request, for a specific error instance, that additional state information be transmitted back from the customer to aid in the debug effort.

Once the data has been delivered to the development team, it must be made available in a form that can be used quickly and easily by all engineers on the team. Watson achieves this through a web-based query page. As an example, in Figure 1 we are requesting a report on all issues experienced by the core SQL Server engine (sqlservr.exe). In this report, a list of issues is returned sorted by number of times that particular issue has been reported. Any of these issues can be double clicked to see more detail and example of which is shown in Figure 2.

From the details display shown in Figure 2, we can see that this error condition has ten instances reported. The issue shown in the example appears to be a problem in the SQL Server Metadata Manager. In many cases, the data available from this report, or from the additional data available via live links, are sufficient to locate and fix a problem without deeper investigation or needing to request more detailed data.

4. Data Collection Agent

In the previous section, we explained how we have developed and deployed Watson support in all retail copies of SQL Server 2000 Service Pack Three. Early experience suggests that Watson is very effective at rapidly feeding back to development teams the top N problems experienced in real operational environments. However Watson only reports problems and not the conditions that lead to the problems or detailed tracking information on how the system was performing when there were no problems. To address this we have developed an ASAF system, currently in limited operational-deployment, that tracks not only failures but uptime, downtime, and numerous other metrics of system activity. Our goals for this work are to obtain actual customer-experienced uptime, learn the causes of system downtime, and both drive and track release-to-release availability improvements while reducing customer administration and product support costs. In the longer term, we hope to find correlations between systems states and subsequent failures, allowing proactive failure prediction and recovery.

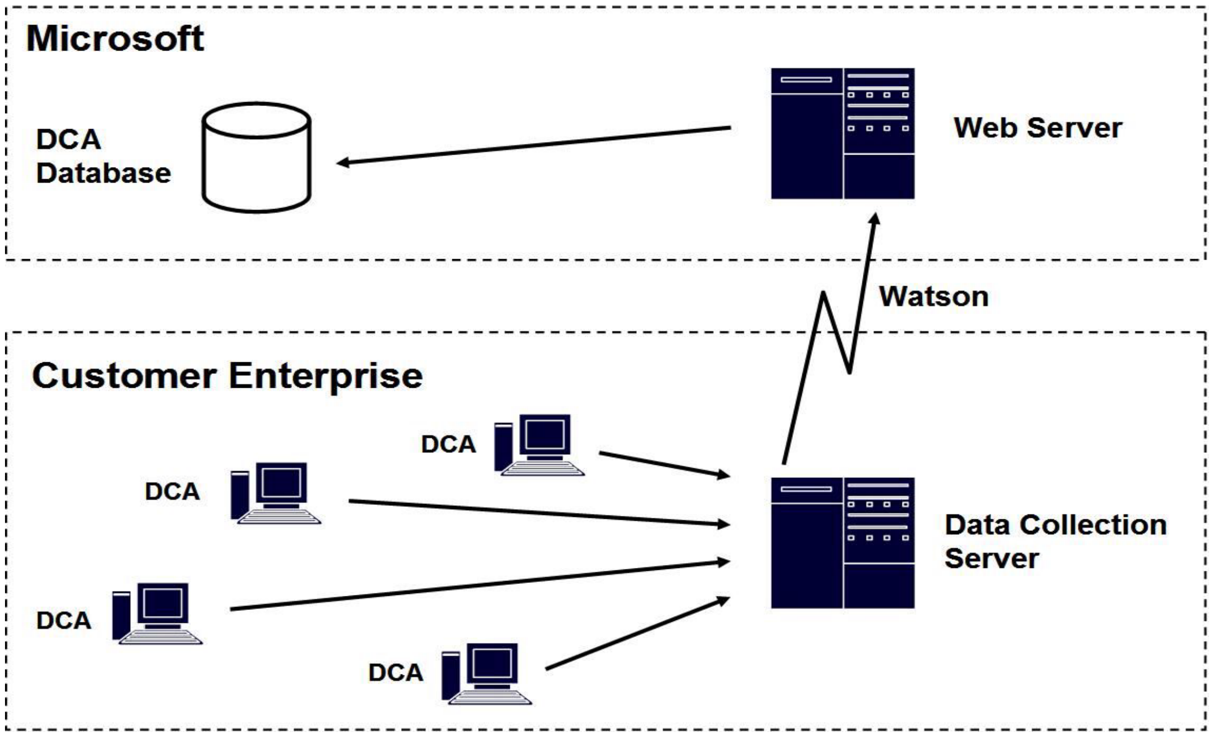


Figure 3: Data Collection Agent Architecture

The Data Collection Agent (DCA) is written as a four-tier system (Figure 3). Data is collected by an agent running on each system being monitored. That data is sent to a central data collection server, of which there needs to be at least one in each customer enterprise. The data is aggregated at the Data Collection Server and then sent using the Watson infrastructure up to the Watson web farm. From there the data is loaded into a SQL Server database for further analysis.

The data collected on each server is divided into three broad classes. The first class is a start-up snap-shot that includes:

- Operating system version and service level
- Database version and service level
- Syscurconfigs table
- SQL server log files and error dump files
- SQL Server trace flags
- OEM system ID
- Number of processors
- Processor Type
- Active processor mask
- % memory in use
- Total physical memory
- Free physical memory
- Total page file size
- Free page file size
- Total virtual memory
- Free virtual memory
- Disk info – Total & available space
- WINNT cluster name if shared disk cluster

The second class is made up of SQL Server state information, including:

- SQL Server trace flags
- Sysperfinfo table
- Sysprocesses table
- Syslocks table
- SQL Server response time
- SQL server specific performance counters:
    - \\SQLServer:Cache Manager(Adhoc SQL Plans)\\Cache Hit Ratio
    - \\SQLServer:Cache Manager(Misc. Normalized Trees)\\Cache Hit Ratio"
    - \\SQLServer:Cache Manager(Prepared SQL Plans)\\Cache Hit Ratio
    - \\SQLServer:Cache Manager(Procedure Plans)\\Cache Hit Ratio
    - \\SQLServer:Cache Manager(Replication Procedure Plan)\\CacheHitRatio
    - \\SQLServer:Cache Manager(Trigger Plans)\\Cache Hit Ratio
    - \\SQLServer:General Statistics\\User Connections

The third and final class of data is comprised of operating system state which includes:

- Application and system event logs
- Select OS performance counters:
- \\Memory\\Available Bytes
- \\PhysicalDisk(_Total)\\% Disk Time
- \\PhysicalDisk(_Total)\\Avg. Disk sec/Read
- \\PhysicalDisk(_Total)\\Avg. Disk sec/Write
- \\PhysicalDisk(_Total)\\Current Disk Queue length
- \\PhysicalDisk(_Total)\\Disk Reads/sec
- \\PhysicalDisk(_Total)\\Disk Writes/sec
- \\Processor(_Total)\\% Processor Time
- \\Processor(_Total)\\Processor Queue length
- \\Server\\Server Sessions
- \\System\\File Read Operations/sec
- \\System\\File Write Operations/sec
- \\System\\Procesor Queue Length

On system start-up, DCA takes the initiation snap-shot (class one above) and then, every minute during normal operation, it takes a snap-shot of the SQL and O/S state (classes two and three above). Under normal operating conditions, every fifth of the once-per-minute collections are returned. But, in the event of a system failure, we send all one minute snap-shots over the last ten minutes. This means we have five minute granularity during standard system operation but one minute granularity snap-shots during the ten minutes prior to a system failure.

4.1 DCA Results

The purpose of the fine grained DCA data is manifold. It gives an accurate measure of product availability in actual customer use. And, during product beta release cycles, the data returned can be used to set specific goals on product availability improvements and help the development team understand exactly what is being achieved and whether further work is needed prior to release. This may sound like a minor achievement but, when an attribute can not be measure accurately, it is very difficult to drive substantial and sustained improvement. Further, since many customer organizations do not have accurate and comparable data on system uptime, it is difficult to reliably get this availability data directly from them.

Through this project we have learned that administrative procedures and experience levels have a substantial impact on the availability achieved, which implies that test systems do not accurately model customer usage, due to substantially different administrative models, training, and responsibilities.

In addition to tracking achieved system availability, we have fine-grained data on system state changes and how they correlate with subsequent system failures. From this we have learned some not particularly surprising results, such as finding that systems are much more likely to fail after new software has been installed than multiple months after installation. That particular finding was expected but other less obvious scenarios can also be evaluated. For example, systems with a large number of locks held (live locks or non-detected deadlocks) frequently get hard cycled as do systems with very high CPU loads. The implication from this correlation is that, when a system is performing poorly, it is likely that an administrator will simply restart the system. We could partially address this issue by providing better diagnostic tools so that a restart would be unnecessary, and we could work on reducing restart times when systems are cycled. Simple steps that can be taken to reduce restart times when a system cycle is probable include check-pointing system state. We believe that simple and non-obvious actions such as a well-placed checkpoint can improve availability by reducing the downtime caused by a future administrative action that we can predict to be likely. Relationships between metrics such as these are very hard to find on test systems, and really only show up reliably in actual customer-administered environments.

The DCA system is still in an early research phase and, although it is deployed on over 120 servers at this time, it is still a long way from deployment in the standard retail version of SQL Server as we have done with Watson. However, even as an immature system, DCA has already conclusively shown many correlations that we had always suspected to exist from experience working with customers and, in a few cases, we have found unexpected correlations:

- There is a high correlation between operating system reboot and improper database shutdown. Upon closer investigation, we have learned that the Windows Service Control Manager does not always allow sufficient time for very large database systems to fully checkpoint on shutdown, vastly increasing recovery time and unnecessarily increasing downtime duration. This is a good example of a small issue that is fairly easy to fix that can have a substantial impact on overall database availability, in that fully 5% of the unclean shutdowns that we have tracked were contributed by this issue and, each time it occurs, the potential downtime contributed can run into the tens of minutes.
- When looking at all instances of lost availability, we saw that fully 66% are shutdowns initiated by the system administrator. This is a higher number than found in past studies and it could be partly contributed to by the fact that these systems under monitoring are running pre-release beta software, although they are fully operational production systems. We need to gather more data on this trend.

- Software upgrades of both the O/S and database system contribute significantly to instances of system shutdowns, with 19% of the clean shutdowns and 5% of the unclean shutdowns following software upgrades (again the contribution of running beta software may artificially increase this result).
- System reboots tend to occur in groups of more than one, which is to say that a single reboot is an excellent predictor of another being highly likely.
- Database management system hard failure is not a significant contributor to lost system availability, with this factor disappearing into the statistical noise. As we get more data, we will be able to more precisely assess the contribution of DBMS failure, but it appears at this point to only be a minor factor, which is to say that other factors contributing to downtime represent much higher improvement leverage.
- 10% of the clean shutdowns were instances of the administrator entering single user mode. Typically this is done to upgrade software, perform offline utility operations, perform schema maintenance, and sometimes for testing purposes. More investigation is required to better understand the primary factors driving the decision to shift to single user mode.
- When looking at the data closely, we see many minor instances of failures of related subsystems and components that do not appear to have received any administrative attention, indicating that it is quite likely that the administrators were unaware of these issues. We believe that many hard failures are the results of several errors accumulating and, if this trend is established, then one action we can take as systems providers to mitigate this is to make it more obvious when a component or subsystem needs administrative attention. By making issues clear and recommending corrective action, we may be able to avoid subsequent hard failures and lost system availability.

4.2 Future Work

In addition to helping to better understand the causes of system downtime, the DCA monitoring system has potential in other applications as well. Some that we are considering as potential future area of investigation:

- *Database feature usage*: which features are actually used by a broad cross section of customers and which are of little value? This information can help drive development investment and help reduce the near infinite accumulation of features in current systems by quantifying how much actual customer use a feature is receiving.
- *Beta release quality assessment:* support to quantify customer usage of product features during the beta program which can help the product team understand the quality and completeness of the beta program and whether more beta customers are required to adequately test the product.
- *Predictive repair or corrective action:* If we are able to reliably predict when the probability of a system failure is high, we can take proactive corrective action. Simple examples include restricting the database request admission policy when resources are scarce and the system is entering an unstable mode. A more radical approach being advocated by Software Rejuvenation researchers is to schedule a system reboot on the assumption that aggressive, yet planned, action will yield better system availability than simply waiting for the expected failure. Although the users will still experience a downtime, we can forewarn the administrators and reduce the duration of system unavailability by properly shutting down the system.

Our Data Collection Agent work will continue, but early results are both confirming some of what experienced database developers would expect and, in some cases, we are finding issues where we didn't predict the impact or frequency of the failure. Overall, a deeper understanding of the causes of downtime is being achieved and, with that better understanding, comes an improved ability to act.

5. Summary

In this work, we have argued that the software development process is inefficient at translating customer experience, especially negative experience, into product improvements. In essence, we are arguing that lack of knowledge across the industry is restricting progress on some of the most important requirements, most notably, system availability. We believe that Active Server Availability Feedback is an effective means to gather a deeper and more detailed understanding of how systems are being used, what issues are encountered in normal customer usage, and what issues, be they product or administrative, are causing downtime and to quantify these contributions. Two Active Server Availability Feedback systems were presented; one now in production use in a commercial DBMS and another that is still in an early research phase. Results were presented for both.

Acknowledgements

I would like to thank David Patterson and the University of California at Berkeley Recovery Oriented Computing team for many ideas and suggestions on software introspection and software availability monitoring that helped initiate this work. And, thanks to Brendan Murphy for his related work on VAX and Windows downtime tracking systems and his continuing contribution and support of the DCA project.

The Office Watson development team conceived, designed, implemented, and continues to support the Watson framework and Steve Lindell, of the SQL Server team, implemented the SQL Server Watson support. Aakash Kambuj wrote the client and mid-tier DCA implementations. Grigory Pogulsky implemented the DCA server-side data acquisition system and manages several hundred gigabytes of database-resident DCA data. Christian Konig, Grigory Pogulsky, Robert Dorr, and Brendan Murphy all contribute greatly to the continuing DCA data analysis effort.

References

[1] J. Gray. "Why Do Computers Stop and What Can Be Done About It?" *Symposium on Reliability in Distributed Software and Database Systems*, 3-2, 1986.

[2] J. Gray. "A Census of Tandem System Availability between 1985 and 1990," *Tandem Technical Report 90.1*, Jan. 1990.

[3] R. Lemos. "Old Code in Windows is a Security Threat", *CNET news.com*, http://news.com.com/2100-1001-934363.html, June 10, 2002.

[4] S. Mourad and D. Andrews. "On the Reliability of the IBM MVS/XA Operating System," *IEEE Transactions on Software Engineering*, Oct. 1987.

[5] B. Murphy and B. Levidow. "Windows 2000 Dependability," *Microsoft Research Technical Report*, MSR-TR-2000-56, June 2000.

[6] B. Murphy and T. Gent. "Measuring System and Software Reliability using an Automated Data Collection Process," *Quality and Reliability Engineering International*, 11:341-353, 1995.

[7] D. Oppenheimer. "Why Do Internet Services Fail and What Can Be Done About It?" *University of California at Berkeley Masters Report*, 2002.

[8] D. Oppenheimer and D. Patterson. "Studying and Using Failure Data from Large-Scale Internet Services," *10th ACM SIGOPS European Workshop*, Saint-Emilion, France, Sept. 2002.

[9] Private correspondence with SAP.

[10] P. Samuelson. "Proof that Properly Anticipated Prices Fluctuate Randomly, *Industrial Management Review*, Vol. 6, 1965. pp.41-49.

[11] M. Sullivan and R. Chillarege. "Software Defects and their Impact on Systems Availability: A Study of Field Failures in Operating Systems," *FTCS 1991*: 2-9.

[12] Watson Development Team. "Reporting Office Application Crashes" http://www.microsoft.com/office/ork/xp/two/admA05.htm.

[13] J. Zbigniew and R. Iyer. "Networked Windows NT Systems Field Failure Data Analysis," *Proc. of IEEE Pacific Rim Intl' Symp. On Dependable Computing (PRDC),* Hong Kong, China, Dec. 1999.