Short summary of Windows Error Reporting system (WER) based on http://research.microsoft.com/apps/pubs/default.aspx?id=81176.
It is based on citations from the original article and shows a general idea behind the system.

WER serves a slightly different purpose than our project. Its main goal is to help debugging Microsoft products by gathering error data from users.

WER aggregates error reports that are likely caused by the same bug into buckets. The goal of the bucketing algorithm is to maintain a property: one bug per bucket and one bucket per bug. To achieve this there are two stages of the bucketing:

- *labelling* - happens on users machine, errors are labeled (assigned to buckets) based on basic data avaliable at the client. Its goal is to find the general cause of the error.
- *classifying* - happens on WER service, errors are placed in new buckets based on further crash data analysis. Its goal is to analyze the labeled error data more deeply and find a specific cause of the problem.

When an error occurs on user machine, client code automatically collects information and creates an error report. Basic report consists only of bucket identifier.
If a solution to the problem is already known, WER provides the client with URL to the solution. If additional data is needed, WER collects a *minidump* (a small stack and memory dump and the configuration of the faulting system). If further data is required, WER can collect full memory dump, memory dumps from related programs, related files or other data queried from the reporting system.

WER enables *statistics-based-debugging* - all error data is stored in a single database so programmers can mine the database to improve debugging. Programmers can sort the buckets and debug the bucket with most report, can find a function that occurs in most buckets and debug that function. It also helps with finding causes which are not immediately obvious from memory dumps.

Short summary of bucketing algorithms

Algorithms are based on collection of hueristics. *Expanding* heuristics increase the number of buckets, *condensing* heuristics decrease the number of buckets. Expanding heuristics should not create new buckets for the same bug and condensing heuristics should not put two different bugs into one bucket.
The idea is to classify the records as well as possible in order to save programmers time and maximize their effectiveness in debugging.

Client-Side Bucketing (Labeling)

	Heuristic	Impact	Description
L1	program_name	Expanding	Include program name in bucket label.
L2	program_version	Expanding	Include program version.
L3	program_timestamp	Expanding	Include program binary timestamp.
L4	module_name	Expanding	Include faulting module name in label.
L5	module_version	Expanding	Include faulting module version.
L6	module_timestamp	Expanding	Include faulting module binary timestamp.
L7	module_offset	Expanding	Include offset of crashing instruction in fault module.
L8	exception_code	Expanding	Cause of unhandled exception.
L9	bugcheck_code	Expanding	Cause of system crash (kernel only)
L10	hang_wait_chain	Expanding	On hang, walk chain of threads waiting on synchronization objects to find root.
L11	setup_product_name	Expanding	For "setup.exe", include product name from version information resource.
L12	pc_on_stack	Condensing	Code was running on stack, remove module offset.
L13	unloaded_module	Condensing	Call or return into memory where a previously loaded module has unloaded.
L14	custom_parameters	Expanding	Additional parameters generated by application-specific client code.
L15	assert_tags*	Condensing	Replace module information with unique in-code assert ID.
L16	timer_asserts*	Expanding	Create a non-crashing error report by in-code ID if user scenario takes too long.
L17	shipping_assert*	Expanding	Create a non-crashing error report if non-fatal invariant is violated
L18	installer_failure	Expanding	Include target application, version, and reason for Windows installer failure.

Figure 4. Top Heuristics for Labeling (run on client).
*Asserts require custom code and/or metadata in each program.

It is run on the client when an error report is generated. The goal is to produce an unique label based on local information that is likely to align with other reports caused by the same bug. In most cases, the only data sent to WER servers is a bucket label.
Primary labeling heuristics generate a bucket label from faulting program, module and offset of the program counter within the module.

For example, user-mode crashed are classified according to the parameters:

- application name
- apllication version
- module name
- module version
- offset into module

Additional heuristics are generated for example when an error is caused by unhandled program exception.
Most of the labeling heuristcs are expanding heuristics intended to put separate bugs into distinct buckets. For example, the hang_wait_chain (L10) heuristic walks the cain of threads waiting for synchronization objects held by threads, starting from the user-input thread. If a root thread is found, the error is report as a hang originating with root thread.

The few condensing heuristics were derived emipirically from common cases when a single bug produced many buckets. For example, the unloaded_module (L13) heuristic condenses all errors where a module has been unloaded prematurely due to a reference counting bug.

Server-Side Bucketing (Classifying)

	Heuristic	Impact	Description
C1	find_correct_stack	Expanding	Walk data structures for known routines to find trap frames, etc. to stack.
C2	skip_core_modules	Expanding	De-prioritize kernel code or core OS user-mode code.
C3	skip_core_drivers	Expanding	De-prioritize first-party drivers for other causes.
C4	skip_library_funcs	Expanding	Skip stack frames containing common functions like memcpy, printf, etc.
C5	third_party	Condensing	Identify third-party code on stack.
C6	function_name	Condensing	Replace module offset with function name.
C7	function_offset	Expanding	Include PC offset in function.
C8	one_bit_corrupt	Condensing	Single-bit errors in code from dump compared to archive copy.
C9	image_corrupt	Condensing	Multi-word errors in code from dump compared to archive copy.
C10	pc_misaligned	Condensing	PC isn't aligned to an instruction.
C11	malware_identified	Condensing	Contains known malware.
C12	old_image	Condensing	Known out-of-date program.
C13	pool_corruptor	Condensing	Known program that severely corrupts heap.
C14	bad_device	Condensing	Identify known faulty devices.
C15	over_clocked_cpu	Condensing	Identify over-clocked CPU.
C16	heap_corruption	Condensing	Heap function failed with corrupt heap.
C17	exception_subcodes	Expanding	Separate invalid-pointer read from write, etc.
C18	custom_extensions	Expanding	Output of registered third-party WER plug-in invoked based target program

Figure 5. Top Heuristics for Classifying (run on server).

The server-side bucketing heuristics are codified in !analyze (an extenstion to Windows Debugger). There were about 500 heuristics dervied empirically.
The most important classifying heuristics (C1 - C5) are a part of an algorithm that analyzes the memory dump to determine which thread context and stack frame most likely caused the error.
There is a number of heuristics to filter out error reports that are unlikely to be debugged (e.g bad memory, misdirected DMA, DMA from a faulty device).
As an another example, kernel dumps are tagged if they contain evidence of known root kits (C11), out-of-date-drivers (C12), drivers known to corrupt the kernel heap (C13) or hardware known to cause memory or computational erros (C14 and C15).