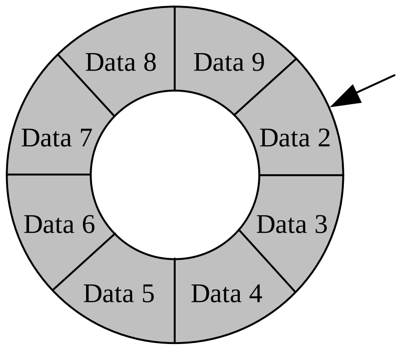
Using Ring Buffer Logging to Help Find Bugs

Brian Marick
marick@visibleworkings.com
www.visibleworkings.com

This paper contains a set of related patterns. All the patterns are concerned with how you, a programmer, get the information you need to understand and fix a software bug. The solutions revolve around the use of a **ring buffer** to log key events in the life of a system**.**



A ring buffer is a buffer with a fixed size. When it fills up, adding another element overwrites the first. The picture to the right shows a ring buffer that can hold eight pieces of data. Nine have been added. The ninth overwrote the first. When another is added, it will overwrite the second.

These patterns have two themes:

1. Be considerate of the users.
2. Over time, evolve a system's logging to make debugging efficient.

Problem	Solution	Pattern Name
How do you make it likely that a user will be able to provide useful information after a failure, *without* inconveniencing her if a failure doesn't happen?	Store a history of the execution of the system in a ring buffer.	Logging Ring Buffer (p. 2)
Users notice problems long after they've happened.	If the program notices odd states, it should dump the log.	Dump Ring Buffer on Error (p. 4)
Users don't trust you.	Allow them to make a reasoned decision about whether to send you a log.	User Friendly Log (p. 6)
How do you know what information to log?	Until you have a better idea, log interface crossings.	Border Patrol (p. 8)
	Log whatever would have made the last bug easy to find	Termite Trails (p. 10)
The buffer always fills up too soon.	Have two logs at different levels of detail.	Two Destinations (p. 12)

Logging Ring Buffer

Customer support refers a customer to you. She's mad. Your program crashed. You ask, "Do you remember the last few things you did before the problem happened?" She does not.

In response, you add logging to the product. Key events and state changes are written to a disk file. You tell customer support that they should ask customers to email the log file when they call in with problems.

Time passes. You are surprised that bug reports from the field often do not contain logs. You ask customer service why. They say that the word of mouth among the user community is that turning off logging is a big performance boost.

More time passes. The project's bug triage team directs you to remove your logging from the product. At one of the key customer sites, your log filled up the hard disk. Being down overnight cost them roughly six times what they paid for your product.

Problem

How do you make it likely that a user will be able to provide useful information after a failure, *without* inconveniencing her if a failure doesn't happen?

Context

These patterns are best suited to standalone applications running on a customer's computer. Some of them do not apply when most of the processing is done on a remote server. Distributed applications need additional patterns.

Forces

- A user's report of what she did before noticing a problem will be inaccurate and incomplete.
- Even an accurate and complete user report does not give all relevant information. Input to a program comes from places other than the user's keyboard and mouse.
- Even if you can examine the frozen state of the program at the moment the problem was noticed, important information is missing. You know the current values of all variables, but not their past values. A stack trace contains some of the past execution history, but not all of it.
- Logging to disk every action that the program takes would be expensive. It would slow the program down. It would consume disk space.
- Users get annoyed if they notice the log when they're not having problems.

Therefore,

Log key events, including user actions, inside a ring buffer. Provide a way for the ring buffer to be written to disk (and thereafter attached to an email message). This is called "dumping the log". Allow the user to manually dump the log when she knows she's found a problem. Make it possible for the program to find its own problems and dump the log itself (see **DUMP RING BUFFER ON ERROR**, p. 4). If needed, provide a tool to recover the log from the remnants of a crashed program (e.g., on Unix, from a core file).

Examples

Linux ring buffers are dumped with the command dmesg(8) [Linux]. The following log was created during startup:

```
Console: 16 point font, 400 scans
Console: colour VGA+ 80x25, 1 virtual console (max 63)
pcibios_init : BIOS32 Service Directory structure at 0x000f73d0
pcibios_init : BIOS32 Service Directory entry at 0xfd7a0
pcibios_init : PCI BIOS revision 2.10 entry at 0xfd9b6
Probing PCI hardware.
Warning : Unknown PCI device (8086:7180).  Please read include/linux/pci.h
Warning : Unknown PCI device (8086:7181).  Please read include/linux/pci.h
Calibrating delay loop.. ok - 333.41 BogoMIPS
Memory: 31140k/32704k available (504k kernel code, 384k reserved, 676k data)
```

The Linux ring buffer is by default 8192 bytes long.

Trace.java [Marick97] is an open source Java tracing subsystem that stores up to 500 trace (log) messages in a list. When the 501st message is added, the first one is dropped and presumably garbage-collected.[1] When dumped, the ring buffer will be a series of messages of this form:

```
=== 1997/09/22 13:52:09 (Controller.event:Controller.java:26) USE
gui: linedraw selected
```

Transarc's **Encina** distributed transaction processing system [Transarc97] uses a 64K trace buffer. Messages stored in the trace buffer can be dumped to a file upon administrative command, when a particular trace message is posted to the log, when a process exits, or when a process is sent a signal.

Consequences

The user can now provide you with more useful information upon a failure.

What information will be useful? See **BORDER PATROL** (p. 8), **TERMITE TRAILS** (p. 10), and **TWO DESTINATIONS** (p. 12).

Will the user do the right thing with the information you've recorded? See **DUMP RING BUFFER ON ERROR** (p. 4) and **USER FRIENDLY LOG** (p. 6).

[1] Note that, in contrast to Linux, there is not a fixed amount of memory allocated to the buffer. A fixed number of messages is easier to implement and has no bad consequences in a large-memory environment where trace messages are likely to mostly be about the same size.

Dump Ring Buffer on Error

You receive a bug report and log from the user. You're saddened to discover that the log has no useful information. The reason? The user fiddled around with the program, trying to recover from the error, before giving up and calling customer support, who told her to dump the log.

Problem

The moment at which a user decides to produce a log may be long after the problem was noticed (or could have been noticed). Crucial information can be lost.

Forces

- Messages that appear in the log after a failure occurs are almost always less useful than messages from before the failure.
- The log is of some finite size, so some messages from before the failure will be lost, possibly including *the* important message.[2]
- If users manually dump the log, they're unlikely to do it immediately.

Therefore,

Whenever it is possible for the program to know it is a good time to dump the log, it should, without waiting for the user.

If the program is dumping the log because it's detected a bug, you might also want to inform the user of the problem and of where to find the log file. That's trickier than it sounds:

- Sometimes the user wouldn't have noticed the bug as such. For example, it might only lead to a weird but temporary screen artifact, or to nothing happening until the user gets impatient and presses a key again. The notification about the bug is actually more disruptive than the bug itself.
- Repeated notifications about the same bug are intensely annoying.

So do this with care. Either do not repeat notifications or aggregate them. For example, Linux and other versions of the Unix kernel will print "Last message repeated 343 times" instead of showing all 343 instances.

Note also that part of the motivation for ring buffers in the first place is to conserve disk space. If the program goes truly haywire, you may end up dumping a largeish ring buffer to disk many times a second, which defeats the purpose of having it.

[2] The size of the ring buffer can often be changed at launch time or even runtime. But that doesn't help with a message that's already been lost.

Examples

In the product for which **Trace.java** was originally designed, there was a top-level exception handler that caught all exceptions unhandled by code lower on the stack. When that top-level code caught an unhandled exception, it dumped the log. So events leading up to errors like array overflows were logged. (Before dumping the log, the handler logged a stack backtrace, so the exact location of the error was known.)

Programmers were also encouraged to put assertions in their code. The assertion routines threw exceptions that would trigger a dump of the log, using the mechanism described above.

In the **Encina** tracing facility, logs can be dumped when a process shuts down (presumably unexpectedly). They can also be dumped when a particular message (such as an error message) is logged. That's useful when a user encounters a failure that can't be reproduced at will. Turning on log dumping for that failure's error message can be used to get more information the next time the problem occurs.

Consequences

You will still sometimes rely on the user to dump the ring buffer, since the program won't always dump it when it should.

You must use "social engineering" to persuade the user to send you the log. See **USER FRIENDLY LOG** (p. 6) for one technique.

User Friendly Log

You add a ring buffer to the product and eagerly await the first bug report with a useful attachment. No one sends one.

So you add a feature: when the program **DUMPS RING BUFFER ON ERROR** (p. 4), it dumps the log across the internet to your site, rather than to a disk on the user's computer. You are fired after a customer service person leaks the URLs browsed by the Prime Minister.

Problem

How do you persuade customers to send you the log?

Forces

- You work in an industry that is doing everything it can to convince customers that it has no respect for their privacy. They don't trust you.
- Automatically retrieving a log is unacceptable, especially since you have no control over what other programmers might choose to log.
- Users would like control over what they're telling you.
- Much internal program data is cryptic. How will the user be able to interpret it?
- Having to review a complicated log is just another way computers are too hard to use.

Therefore,

Make log messages understandable by a user. There should be no cryptic data (such as long strings of numbers that might be encoded text). It is best if the log is formatted so that it's easily scannable, so that the reader can quickly reassure herself that nothing looks suspicious. Text that is unpleasant to look at won't be – and the log will never be sent in. If there is baroque content (such as Java package names, filenames, and line numbers), explain the format of a message in a place where the user is likely to find it.

Encourage programmers to think about privacy when writing log messages. For example, there is no reason ever to log a password in plaintext. Logging email addresses might be avoided. Remember that a main purpose of logs is to record things the user has likely forgotten. Her email address won't be one of those. URLs would probably have to be logged – it's too likely that the user won't know them if she clicked from place to place – but that's OK if the user can review them before the bug report arrives.

Examples

Netscape's **Quality Feedback Agent** [Netscape] pops up when the Netscape browser crashes. It sends information about the crash through a direct connection to Netscape (rather than asking you to send a log by email). It gives you a chance to review the data before sending it, and the documentation clearly states “Sensitive information such as web sites visited, email messages or addresses, passwords, and profiles will not be collected.” However, at least one person I know doesn’t trust it. He would rather send a log as an email attachment than trust a program to send only what it claims it’s sending. This perhaps seems excessively paranoid, but our opinions about what users should think are not relevant.

I don’t know if this person noticed that you can save the Agent’s data to a file. However, when I saved it, it was 187,425 characters long, including hex and ascii stack dumps, program instructions, process lists, device driver lists, DLL lists, and so forth. Things like this:

```
[ C70] 00 00 09 01 00 00 87 3E 00 00 A8 49 00 00 2D 00 [.......>...I.-.]
[ C80] 00 00 65 6D 61 63 73 2E 65 78 65 00 70 A3 20 EA [..emacs.exe.p. .]
[ C90] B9 D0 BF 01 00 70 F4 0E 00 60 F4 0E 42 05 00 00 [.....p...`..B...]
```

That may be too long, too inclusive, or too cryptic for a suspicious user.

The **Trace.java** output goes to some lengths to make sure that the data the programmer provided fits on one line, in the author’s perhaps-misguided belief that people find odd line breaks offputting:

```
=== 1997/09/22 13:52:09 (Controller.event:Controller.java:26) USE
gui: linedraw selected
```

Encina’s trace examples show text wrapping across lines and more cryptic numbers:

```
1    22753 96/01/01-08:17:51.003038 a0040437 W  Unable to bind to
server /.:/branch1/server/sfsServer1 (DCE-rpc-0214: not registered in
endpoint map)
```

But their audience is hardened system administrators. **Trace.java**’s application’s audience was to be ordinary people. (The product never made it out of beta.)

Resulting Context

The user *still* may not send you the log. However, there’s an ancillary benefit: if the log is readable, the user may be able to solve the problem on her own.

Border Patrol

Context

The system provides a ring buffer and the tools to dump it.

Problem

How do programmers decide what information to log?

Forces

- Too little detail means that vital information will never have been logged.
- Too much detail will increase the chance that vital information will have been overwritten.
- Too much detail also requires programmers to spend time adding logging statements that may never be needed.
- It's hard to know what you'll need until after you need it.
- A history of what the user did is very often useful. They performed at least some of the actions that led to the problem.
- Programmers will use the log information. Individual programmers often have responsibility for particular **subsystems**. For example, one programmer might be responsible for networking, one for the security kernel, and one for the GUI. Subsystems are generally significantly larger than a single class; they are more likely to be their own package.
- Many bugs are localized within a single subsystem. Many others are due to the interaction between two subsystems. Thus, a big part of debugging is giving the bug report and log to the right person.

Therefore,

If you don't have a better idea of what to log, log user interface events. For purposes of most programmers, a UI event is something that provokes messages to other subsystems. Something that happens solely within the UI layer is not of enough general interest to justify its space in the log. For example, most programmers do not want to know that a particular mouse button had been clicked somewhere in a dialog box, but rather that the changes of the whole dialog had been applied.

Next, log calls into a subsystem that cause it to perform significant processing. You probably would not log a getter function that just fetched state. You might well log a setter function that changed subsystem state, especially if that state affected future processing within the subsystem (e.g., if it did something more than just hang around waiting to be gotten).

Ring buffer log entries should be tagged with the name of the subsystem from which they come. This makes it more apparent which programmers' code is relevant. (Testers also often use this information.)

Examples

In the **GNU Emacs** editor [Emacs], the view-lossage function shows a ring buffer that records keypresses, mouse clicks, whether mouse movement occurred (but not to where the mouse moved), and which menu items were selected. Here's a sample:

```
C-x o C-SPC escape > escape w C-g C-h a d r i b b l
e return C-x o C-n C-n C-n C-n C-x o escape x s h e
l l return down-mouse-1 mouse-movement mouse-1 double-down-mouse-1
mouse-1 escape > C-x C-b C-x o C-n C-n C-n f ( l i
s p SPC r u l e s ) C-j escape x v i e w - l o s s
a g e return
```

This function (and the related "dribble file", which can record all key presses) is serviceable in the absence of any other level of logging.

Trace.java allows logging by subsystem. The subsystem name appears in the trace output, allowing easy skimming and filtering. In the product for which it was originally written, the user interface subsystem did not log keypresses and mouse presses. Instead, it mostly logged finished user actions that caused calls into other subsystems. The result was considerably less than one entry per second in the ring buffer, so the default 500-message buffer size was enough to capture the sequence of user actions that led to most bugs.

By default, other subsystems did not log incoming calls. However, there was a mechanism (described under **TWO DESTINATIONS**, p. 12) to ask for more detail in the log. Some, but not all, subsystems would then log cross-subsystem calls (and other events at a similar level of detail). Some of the testers customarily ran the system this way. The logs were still fairly readable, without the need to wade through a lot of irrelevant detail.

Resulting Context

This is an imperfect solution for several reasons.

1. If only interface crossings are logged, vital information will be missing.
2. As the ring buffer fills, useless but newer information will still drive out old and valuable information.
3. Programmers (and users deciding whether to mail in a log) will still have to look at never-relevant information that was logged in the hope that someday it would be useful.

TERMITE TRAILS (p. 10) and **TWO DESTINATIONS** (p. 12) help resolve remaining forces.

Termite Trails

Context

There is some logging in the system, but it's in the wrong place and to the wrong level of detail. You know that you will someday redeploy the system or ship new executables, so improving logging is worth your time.

Problem

How do you improve the logging in the system?

Forces

- The log is used to help debug. Failing to add a useful log message will someday slow down someone's debugging.
- But adding a useless message will just mean more useless information to wade through, and it could push important information off the ring buffer.
- It's folk wisdom that bugs cluster, meaning that the best place to look for more bugs is where you've already found a bunch.
- Deciding what to put in a log message is not always easy. It's hard to predict what will be useful later, especially if it has to be useful to someone else.

Therefore,

Don't try too hard to predict what will be useful later. Think about the last bug you fixed. Was the sequence of events that caused it difficult to reproduce? Did you have trouble localizing the fault, finding roughly where the bad code was? If so, add logging statements that would have made the task easier.

Also log the values of key variables. These variables are those whose different values can cause significantly different execution paths. They will help later programmers understand why the program got to a particular place.

Be reluctant to add logging statements that are specific to the particular bug, rather than to the path to that bug. For example, if the key to the bug is miscalculation of a particular instance variable, you might now be tempted to log its value in some of the places it's calculated or used. But that's likely to be detail not useful for future bugs, and it might push useful detail off the ring buffer.

By analogy, you should be giving directions to the general vicinity of the bug, not door to door directions, since the next bug will live on a different street in the same neighborhood.

While fixing bugs in a subsystem over a long period of time, you may find that certain logging statements are never useful. Consider removing them. Be careful, though: what's not useful to you might be useful to others.

Example

Like many trace packages, **Trace.java** has multiple levels of logging. (See **TWO DESTINATIONS**, p. 12.) “General vicinity” messages can be logged at a higher level, one that appears in a customer’s ring buffer. Messages inspired by the details of a specific bug can go at a Debug level that’s only logged when explicitly turned on. Those messages won’t help with an unrepeatable bug, but they also won’t crowd out the general vicinity messages from the log. They might help with debugging a repeatable bug, and in that context do no harm.

Resulting Context

The system becomes easier to understand. That understanding may be idiosyncratic to one person, the person adding logging. The log messages may not be as helpful to other people. Shifting people among subsystems (as in Extreme Programming [Beck99]) may help. So may discussing logging messages with testers, another constituency that will make heavy use of them.

Two Destinations

Context

You’re logging important information into the ring buffer, which regularly forgets it.

Problem

Beginnings are a special time. Important information about the system’s environment is most naturally discovered and logged at system or subsystem startup. How do you avoid losing that information as the ring buffer fills?

More generally, some information is so likely to have value that you never want to discard it. How can this be done?

Forces

- Some old information must be preserved.
- Judging the future importance of a log message is hard. Should an old but important log message take precedence over a new but less important one? Who knows?
- It’s confusing to have to look for information in more than one place.

Therefore,

Provide several levels of logging. Each successive level records more detail and leads to more log entries. For example, a “milestone” log entry might record a major event in the system’s life (such as a subsystem’s initialization or shutdown, or the establishment of a connection that’s expected to persist for hours). Milestones happen infrequently. The lower priority “usage” log entries would record significant user interface events (like applying a dialog). They happen more often. Lower than the “usage” level would be the “event” level, which is used to record subsystem interface crossings. The event level is lower than usage because a single UI event might cause a cascade of interface crossings and event entries.

Provide two logs. One is the ring buffer. The other is permanent (disk) storage.

Messages above a certain priority level go to *both* the ring buffer and permanent log. They go into the ring buffer because people will want to read it sequentially. They won’t want to keep referring to the other log to see if any higher-priority messages happened around the time of a particular ring buffer message.

If the code **DUMPS THE RING BUFFER ON ERROR** (p. 4), it should dump it into the permanent log. That becomes the single file that the user might send you. The same is true if the user manually dumps the log.

To avoid filling up the disk, the permanent log may have a bounded size, but it should be much larger than the ring buffer.

Examples

Trace.java has essentially the structure described above. The priority levels are Error, Warning, World (major system milestone), Usage (UI event, not mouse presses), Event (internal interface crossings), Debug, and Verbose. By default, Error, Warning, and World messages go to both destinations. Those three, plus Usage messages, go to the ring buffer. The remaining messages go nowhere unless the logging level is changed.

One problem with dumping the ring buffer to the disk log is that the entries became out of order. For example, you might have this series of entries:

$$\left.\begin{array}{l}\text{Entry W1 (World level)}\\ \text{Entry W2 (World level)}\\ \text{Entry W3 (World level)}\\ \text{Entry W1 (World level)}\\ \text{Entry E1 (Error level)}\end{array}\right\}\begin{array}{l}\text{Originally}\\ \text{on disk}\end{array}$$

$$\left.\begin{array}{l}\text{Entry U1 (Usage level)}\\ \text{Entry U2 (Usage level)}\\ \text{Entry W2 (World level)}\\ \text{Entry U3 (Usage level)}\\ \text{Entry W3 (World level)}\\ \text{Entry E1 (Error level)}\end{array}\right\}\begin{array}{l}\text{From ring}\\ \text{buffer}\end{array}$$

Putting clear separators ("RING BUFFER LOG BEGINS HERE!", "RING BUFFER LOG ENDS HERE!") between the two types of entries helped, as did timestamping entries, but this was still unsatisfactory. A simple utility to sort entries by timestamp, then remove duplicates, would probably have eventually been necessary had the product made it past early beta.

Encina works much the same way. It has the following trace categories: Audit, Error, Fatal, Entry, Event, Param, Dump. By default, Audit, Error, and Fatal messages go to both a disk file and the console. The other messages go into the ring buffer. These defaults can be changed. Unlike **Trace.java**, Encina's categories are not organized in levels; they're toggled individually.

Resulting Context

There is a single place to look for a description of system problems. It includes high-priority entries at first, more detail at the end.

An Unexplored Variant

By default, **Trace.java** allows an unbounded permanent log, but it can also enforce a limit of N bytes. If the log file grows above N/2 bytes, it is closed, renamed to a backup file, then reopened as an empty file. This is a simple approximation to a ring buffer: in the worst case, you still have N/2 bytes of log to examine (or the complete log, if it never hit the limit).

This scheme loses the potentially valuable startup information. In retrospect, it would have been better to have three files. The first would store the original N/3 bytes of log, and the second two would store up to the most recent 2N/3.

Acknowledgements

I thank my PLoP shepherd, Kyle Brown, for his helpful suggestions. I thank Ralph Johnson for discussion of an earlier and spectacularly unsuccessful version. This paper was workshopped at PLoP 2000; my thanks to Rossana Andrade, Todd Coram, Brian Foote, Terry Fujimo, Robert S. Hanmer, William Opdyke, Carlos O'Ryan, and Juha Pärssinen.

References

[Beck99]

Kent Beck, *Extreme Programming Explained: Embrace Change*, 1999.

[Emacs]

Free Software Foundation, *Gnu Emacs Manual*.
http://www.gnu.org/manual/emacs-20.3/emacs.html

[Linux]

dmesg(8) manpage, http://howto.tucows.com/man/man8/dmesg.8.html

[Marick97]

Brian Marick, “The Trace.java User’s Guide”,
http://www.visibleworkings.com/trace/Documentation/Trace.html

[Netscape]

Netscape Corporation, “Netscape Quality Feedback System for Netscape Communicator 4.5”.
http://home.netscape.com/communicator/navigator/v4.5/qfs1.html

[Transarc97]

Transarc Corporation, *Encina Administration Guide, Volume 1*, 1997.
http://www.transarc.com/Library/documentation/txseries/4.2/aix/en_US/html/aetga1/aetga118.htm