Log Data Management Patterns

Amir Raveh
amirr@netvision.net.il

Hardware: those parts of the system you can kick.
Software: those parts of the system you can merely curse. (anon)

Introduction

Once we deliver the software to our customers, it starts its (hopefully) useful life. And just as any living being or artifact, it seldom ends up working as we intended.

When software fails, we start looking for information that can provide us with hints as to what went wrong, and where. Logging can provide us with the means to make the required information available when we need it. Logs provide support engineers and developers with essential windows into the system, collecting information and presenting it to best allow for analysis and to help pinpoint the source of problems that may arise. Logs can also provide us with information when no failure is visible. Logs are a valuable source of information about how the system is put to use by our customers, system performance and a useful resource for identifying behavior patterns.

Logs seem quite simple and straightforward – just set up a persistent store that will record, or log, events as they occur in the system. Create this persistent store so it indicates the time of each event and relevant information according to the problem or event encountered and the troubleshooting needs you anticipate for this system.

Logs require resources from the system, such as processing power to log an event and the storage space needed to keep the event for future reference. No system can afford to collect logs forever – at some point the system will run out of storage space. Therefore, the system should be designed with a mechanism that will keep logs from overflowing.

The patterns described here provide software designers with a few solutions to problems encountered in log data management in two main categories – *Log Management Policy* and *Log Purging*. These categories provide us with answers to the following questions:

- How should we write the next event to our logs?
- How much of the system's processing power will be consumed by logging?
- How can we keep our logs from overflowing?

- How can we keep log data accessible for system maintenance?
- What effort is needed to find an event when we are trying to resolve a software problem?

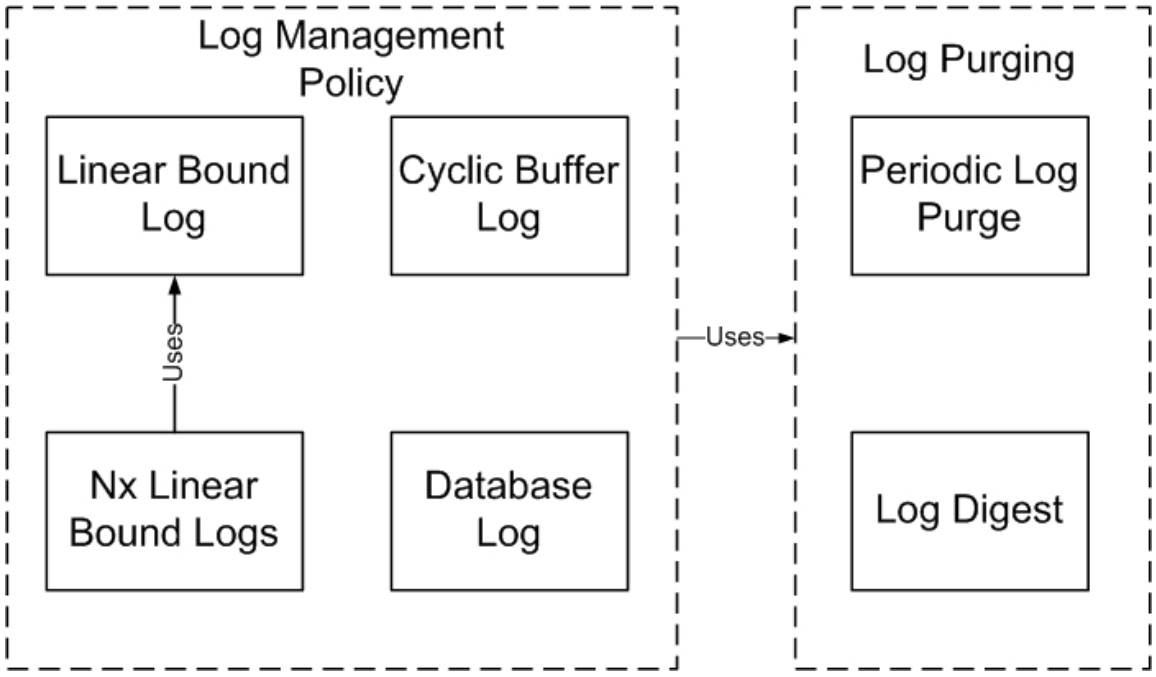


Figure 1 - Log Data Management Patterns

Log management policy patterns present us with four approaches to how we handle writing events to a log and what is the underlying log mechanism, each balancing the forces of this problem in a different manner:

Linear Bound Log presents us with the simplest solution in terms of implementation complexity and processing power per event logged, at the cost of losing all logged events each time log space is recycled.

Cyclic Buffer Log and **Nx Linear Bound Logs** reduce the loss of events at the cost of implementation complexity and processing per event logged. Their use may also make searching for events in a log more difficult.

Database Log uses a database to store events, so logs are very easy to access and purge by using database queries – but requires more processing overhead.

We also provide patterns for *log purging* that can be applied to all of the above log management policies:

Periodic Log Purge is triggered by a scheduling mechanism to purge the logs of their content.

Log Digest is a way to reduce the loss of information during log purges by creating a summary of the events in logs.

These patterns are part of a work in progress on the topic of designing software for maintainability and the work done in maintaining software products[1]. This larger pattern language structure is displayed in Figure 2 (patterns with shaded background are from related work by Weir and Noble[2] and by Harrison[3]).

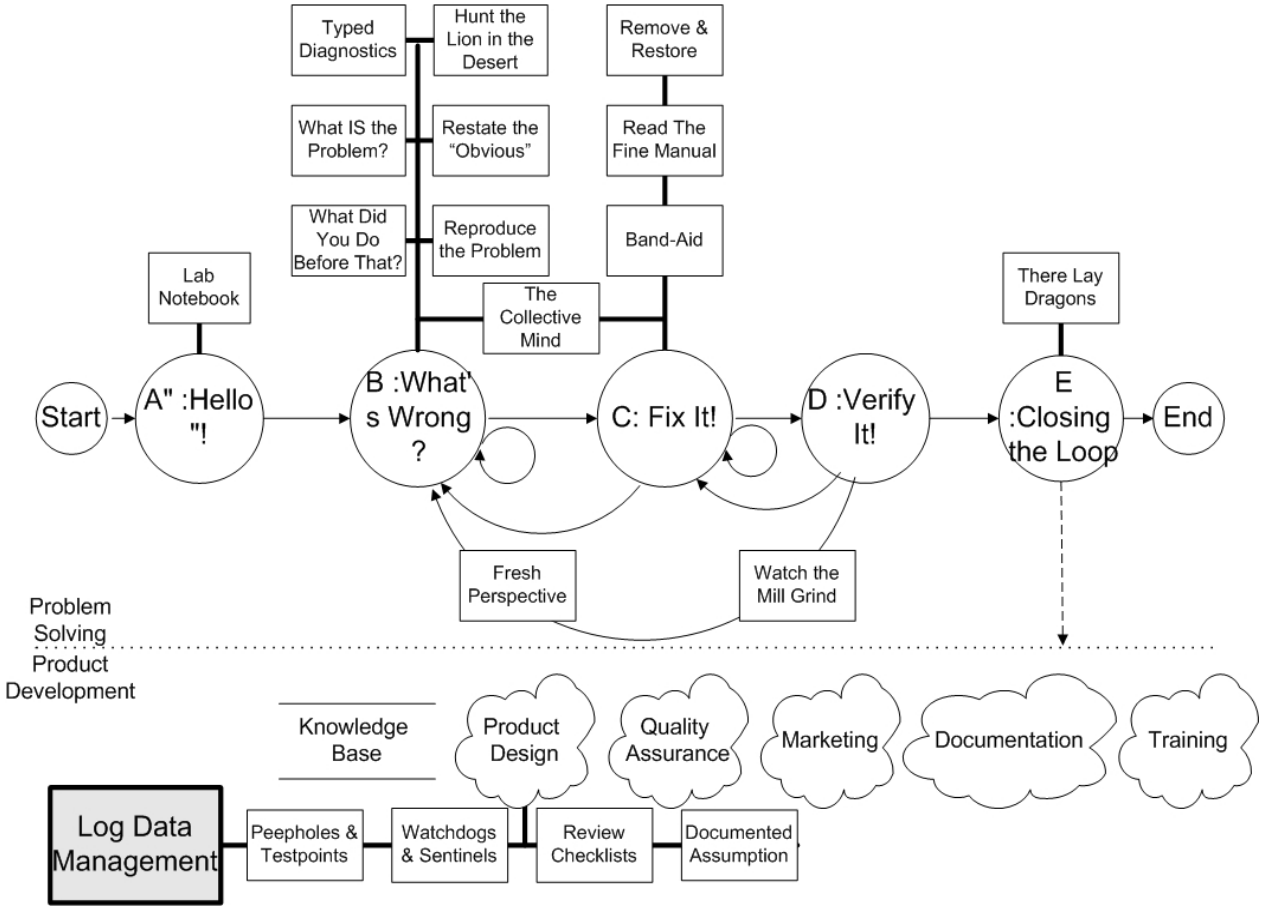
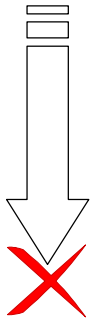


Figure 2 - Design for Maintainability, Maintainability by Design

Pattern Language Structure



Linear Bound Log

Context

You are designing the logging for a software system. You need data of the recent events in the system, but while storage is limited, the arrival rate of events cannot be foreseen. Still, you are required to provide a mechanism for logging events so that the data collected should contain enough history to allow for resolving problems.

Problem

How can you protect system resources from being consumed by an accumulation of log events?

Forces

- You want to protect your system from damage or loss of persistent storage in case the arrival rate of events is higher than expected.
- You want to keep the processing overhead per event logged low.
- You want to keep the logging code as simple as possible.
- You want to keep history of logged events as far as possible into the past.

Solution

Assign the log with a maximal size. Once this size is reached, recycle the log space - the next message will be written to the beginning of the same log, and all previous events are erased.

Resulting Context

The system never goes over the size allocated for the log in persistent storage, saving you from triggering resource overflow by the events you log. However, this also means that once the log reaches its size limit, the system loses all past history of logged events[*].

This side effect can be reduced by using **Cyclic Buffer Log** or **NxLinear Bound Logs**, at the cost of increased implementation complexity and processing overhead. Loss of all

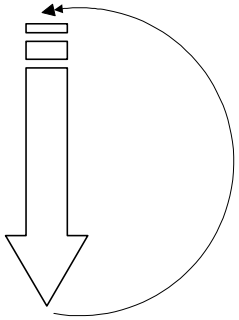
---

[*] Seasoned programmers and support engineers are well aware that even after the contents of the log have been erased and their space is available for reallocarion, they might still be accessible using tools such as file undelete, memory dump and free list walker.

logged data once the pre-allocated size is reached can be reduced by using **Periodic Log Purge** or **Log Digest**.

Known Uses

- Early HTTP servers used a log file with this simple mechanism.
- This pattern is used to construct **NxLinear Bound Logs**.



Cyclic Buffer Log

Context

You are designing the logging for a software system. You need data of the recent events in the system, but while storage is limited, the arrival rate of events cannot be foreseen. Still, you are required to provide a mechanism for logging events so that the data collected should contain enough history to allow for resolving problems. The impacts of losing all data once storage size has been reached in **Linear Bound Log** are too high.

Problem

How can you protect system resources from being consumed by an accumulation of log events?

Forces

- You want to protect your system from damage or loss of persistent storage in case the arrival rate of events is higher than expected.
- You want to keep the processing overhead per event logged low.
- You want to keep the logging code as simple as possible.
- You want to keep history of logged events for as far as possible into the past.

Solution

Estimate the size you can allocate for log messages. Once the log reaches the predetermined size, overwrite the oldest message with the newest, using the allocated log storage size as a cyclic buffer or a finite FIFO.

Resulting Context

You never go over the size allocated for the log persistent storage, saving you from triggering resource overflow by the events you log. You also keep the newest logged events accessible for troubleshooting, losing only the oldest events when new events arrive.

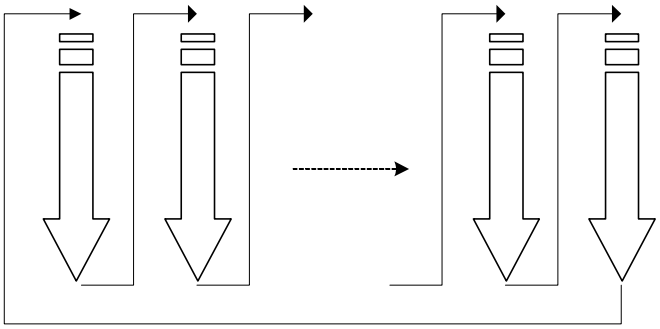
However, in some environments this solution increases the processing overhead required to log an event and the complexity of the code that implements logging. The readability of the log is also impacted, as you now need to scan the log to identify the location of the newest and oldest events, and read sequentially from those points. This side effect can be reduced by using a log browser or introducing a log query system. Loss of logged data once the pre-allocated size is reached can be reduced by using **Periodic Log Purge** or **Log Digest**.

Known Uses

- The Java MemoryHandler class in java.util.logging provides an in-memory cyclic log.
- The Apache log4j[4] the CyclicBuffer provides cyclic buffering for logging.

- In UNIX systems, the syslog(2) uses an in-kernel cyclic memory buffer of size LOG_BUFF_LEN.
- The SwitchMATE™ cellular network management system by Motorola uses cyclic logs in files for its logging.

NxLinear Bound Logs



Context

You are designing the logging for a software system. You need data of the recent events in the system, but while storage is limited, the arrival rate of events cannot be foreseen. Still, you are required to provide a mechanism for logging events so that the data collected should contain enough history to allow for resolving problems.

Problem

How can you protect system resources from being consumed by an accumulation of log events?

Forces

- You want to protect your system from damage or loss of persistent storage in case the arrival rate of events is higher than expected.
- You want to keep the processing overhead per event logged low.
- You want to keep the logging code as simple as possible.
- You want to keep history of logged events for as far as possible into the past.

Solution

Estimate the size you can allocate for log messages, divide it among **N Linear Bound Log**s. Start writing log messages into the first log. Once the log reaches the predetermined size, create a new log. Once you have reached the end of the size allocated to the N-th log, reuse the first log.

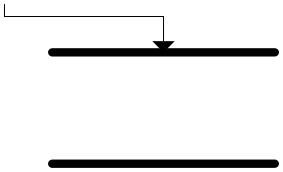
Resulting Context

The system never goes over the size allocated for the log persistent storage, saving you from triggering resource overflow by the events you log. You also keep the newest logged events accessible for troubleshooting, losing only the oldest events in the oldest log when new events arrive.

However, this solution increases in some environments the processing overhead required to log an event and the complexity of the code that implements logging. The readability of the log is also impacted, as you now need to scan all logs to identify the location of the events you are looking for. This side effect can be reduced by using a log browser, introducing a log query system or creating an index of the logs and their content. Loss of logged data once the pre-allocated size is reached can be reduced by using **Periodic Log Purge** or **Log Digest**.

Known Uses

- The Apache log4j[4] the RollingFileAppender provides this mechanism using the methods setMaxBackupIndex(int maxBackups) to set the maximum number of backup files to keep around, and setMaximumFileSize(long maxFileSize) to set the maximum size that the output file is allowed to reach before being rolled over to backup files.
- A much older implementation was the use of N tapes for archiving logs in computer systems using reel to reel tapes. Once a tape ran to its end logging events, it was replaced by reusing the oldest tape.



Database for Logs

Context

You are designing the logging for a software system. You need data of the recent events in the system, but while storage is limited, the arrival rate of events cannot be foreseen. Still, you are required to provide a mechanism for logging events so that the data collected should contain enough history to allow for resolving problems. You are willing to sacrifice computing power and storage efficiency for flexibility of log browsing and purging.

Problem

How can you protect system resources from being consumed by the accumulation of log events?

Forces

- You want to protect your system from damage or loss of persistent storage in case the arrival rate of events is higher than expected.
- You want to keep the processing overhead per event logged low.
- You want to keep the logging code as simple as possible.
- You want to keep history of logged events for as far as possible into the past.

Solution

Log messages into a database. By logging to a database instead of a stream, you also gain random access to single events, as well as easy access to event statistics. You also increase your ability to access logged events, and provide additional flexibility in processing events using existing standard tools such as query languages, browsers and report generators.
The purge mechanism will use database features (such as triggers and queries) to maintain the log under controlled size.

Resulting Context

The system never goes over the size allocated for the log persistent storage, saving you from triggering resource shortage. However, the additional flexibility and the standard tool set offered by databases come at a higher price from system resources such as processing power, memory and storage.
Loss of logged data once the pre-allocated size is reached can be reduced by using **Periodic Log Purge** or **Log Digest**, which may benefit from using standard database features to make purging and digesting the logs more sophisticated.

Known Uses

- The Apache log4j[4] DBAppender provides saving log messages into a database. Supported databases include PostgresSQL, MySQL, Oracle, DB2, MsSQL and HSQL.
- The Motorola MiLOC™ location server uses a database for logging.

Periodic Log Shrink

Context

You are designing the logging for a software system. You need data of the recent events in the system, but while storage is limited, the arrival rate of events cannot be foreseen. Still, you are required to provide a mechanism for logging events so that the data collected should contain enough history to allow for resolving problems – while keeping persistent storage under control.

Problem

How can you keep the size of logs under control, when you have no control over the rate at which log events are generated?

Forces

- You want to protect your system from damage or loss of persistent storage in case the arrival rate of events is higher than expected.
- You want to keep the processing overhead of log data maintenance as low as possible.
- You want to keep the logging code as simple as possible.
- You want to keep history of logged events for as far as possible into the past.

Solution

Provide the logging system with a mechanism to execute a periodic shrink of the logs by deleting events or logs. Use a scheduling mechanism to trigger code to go over the logs at pre-determined schedule and delete events from the logs, delete the logs or move events and logs to secondary storage.

You may also keep the newest logged events accessible for troubleshooting, losing only a pre-determined period of the oldest events in the log.

Resulting Context

You reduce the probability of going over the size allocated for the log persistent storage, saving you from triggering resource overflow by the events you log. However, this solution increases in some environments the processing overhead required by your logging mechanism to log an event and the complexity of the code that implements logging. In some logging implementations purging will halt or slow down the logging of new events.
The readability of the log is also severely impacted, as you lose all events that are purged.
This side effect can be reduced by producing a **Log Digest** before purging, to keep at hand some of the information from the logs to be shrunk.

Bear in mind that purging the files periodically only reduces the probability of going over the allocated size – a surge of events may still happen between two purges, so the system still needs a mechanism that will resolve this problem at the event logging point, such as **Linear Bound Log**, **Cyclic Buffer Log**, **Nx Linear Bound Logs** and **Database Log**.

Known Uses

- Apache log4j[4] provides two such mechanisms:
  - DailyRollingFileAppender, which provides a way to roll over log files on monthly, daily and hourly schedule.
  - ExternallyRolledFileAppender listens on a socket on the port specified by the Port property for a "RollOver" message. When such a message is received, the underlying log file is rolled over and an acknowledgment message is sent back to the process initiating the roll over.
- The Unix System V generates daily archiving for the system accounting records (sar).

Log Digest

Context

You are designing the logging for a software system. You need data of the recent events in the system, but while storage is limited, the arrival rate of events cannot be foreseen. Still, you are required to provide a mechanism for logging events so that the data collected should contain enough history to allow for resolving problems – while keeping persistent storage under control.

Problem

Given finite resources, you need to make a choice between losing new incoming events and deleting old events.

Forces

- You want to protect your system from damage or loss of persistent storage in case the arrival rate of events is higher than expected.
- You want to keep the processing overhead of log data maintenance as low as possible.
- You want to keep the purging code as simple as possible.
- You want to keep history of the important logged events for as far as possible into the past.
- It is difficult to know what is important in advance.

Solution

When the *log management policy* is reusing a log, **have it create a digest of its contents, before they are lost**. The digest can range from a statistical summary of the events to keeping a few selected events (such as the most severe events) all the way to compressing the entire log.

Resulting Context

The system keeps the information that you deem necessary for longer periods. This way you can at least have a hint of what events were recorded by the system before the logs were purged or overwritten. However, this solution may increase the processing overhead required by your logging mechanism to purge logs and the complexity of the code that implements purging.

A digest only keeps information it was programmed in advance to keep – this also means that you might miss the unexpected events that led to the problem you are currently troubleshooting. Modifying the **Log Digest** code might help you collect more information, but might also alter the behavior of your system. Bear in mind that **Log Digest**s also require storage, and plan for their purging as well.

Known Uses

- The Unix System V produces daily log digests of the system accounting records (sir).
- Novell system provides an Automated Volume Purge[5] – AVP creates a digest of logs and sends them by email to the administrator, before purging.
- The SwitchMATE™ cellular network management system by Motorola uses LZW file compression of logs it harvests from network elements. These compressed logs are than kept for the last seven days.

Credits

I would like to thank the customers and colleagues I have worked with designing and maintaining products and systems – it is through their perspectives that I have learned.

Credit is also due to Ofra Homsky, for her continued inspiration, and for her feedback that helps these patterns evolve.

Arno Schmidmeier provided me with valuable feedback and insights towards making a disjoint set of associations into a pattern language.

I thank my PLoP 2005 shepherd, Doug Schmidt for his insights, perspective and feedback – his contributions has helped clarify these patterns (and unify some of the short paragraphs in this document…).

This paper benefited from the caring and the feedback of the members of PLoP 2005 Writers' Workshop 2 – Kanwardeep Singh Ahluwalia, Danny Dig, Bob Hanmer, Sargon Hasso, Timothy C. Lethbridge, Ricardo Jorge Lopez, Hafiz Munawar, Adam Murray, John Sinnot, Leon Welicki and Paul Adamczyk (in absentia) - led by Linda Rising and Dick Gabriel. Viewing the paper through their eyes and minds has helped me improve the patterns and provided me with more insights into this problem domain and I am thankful for their contributions and recommendations.
Any errors, mistakes, imperfections and omissions are mine, though…

References

1 Technical Support patterns, Ofra Homsky and Amir Raveh, Proceedings of EuroPLoP 2003
2 Process Patterns for Personal Practice, Charles Weir & James Noble, Proceedings of EuroPLoP 1999
3 Patterns for Logging Diagnostic Messages, Neil B. Harrison, PLoP 1996
4 http://logging.apache.org/log4j/docs
5 http://www.novell.com/coolsolutions/tools/13636.html