Unified error reporting -- A worthy goal?

Andi Kleen, Intel Corporation
Sep 2009
andi@firstfloor.org
errors

- standardized errors
  - machine checks
  - pci-express errors
- platform errors
  - thermal errors
  - APEI
- storage errors
  - IO errors
  - SMART events
- network errors
  - link lost
- random errors from drivers
  - failover
- software errors
  - out of memory
scope

- concentrating on platform hardware errors for now

- the others possibly later

- but especially software errors are hard
  - because there are so many of them
what can you do with errors:

- log them

- categorize them: display critical ones on the desktop as pop up
- account them, keep statistics
  - that many errors on device X in last 24hours

- trigger events
  - e.g. when more than X errors in 24h call this shell script
  - which pages admin, support, triggers failover
  - or on a small home servers starts blinking the red LED
    - (after all what else is the "LED subsystem" good for?)
audiences

- desktop user
- normal system administrator
- expert
- automated analysis tool
- cluster logging
the desktop user

- don’t really understand errors
  - at best a very high level summary

- should not be unnecessarily concerned
  - needs classification, hiding

- graphical interface

- localization

- details should still be available for expert support
normal system administrator

- largely same as desktop user

- only really needs high level summary

- should not be unnecessarily alarmed

- really wants to identify failed part

- graphical interface not as important
  - can access log files
  - but still useful if not intrusive
  - needs reporting to the console
expert / automatic tools

- compatibility crucial

- still want high level summary
  - but all the details should be available

- interface to other tools
  - might put error from a cluster in central database
so what’s wrong with printk?

- difficult to parse

- good errors are verbose

- printk is traditionally for 1-2 lines
  - most printks with more information are a mess
  - no clear record boundaries

- categorization / severity important

- good errors too verbose for kernel log
what’s good with printk

- it’s the standard
  - a lot of people know where to look

- there are lots of tools to handle it
  - including network servers
  - but often not very good

- should be used for some high level categorization
  - but only those errors that don’t make sense to hide
error metadata

- hardware errors
  - ultimate goal is to identify the failed part
  - various other information

- various other data useful
  - for example dropped event count

- advantage of standard records
  - they tend to be reasonably well documented
  - so you can point sophisticated users to documents
  - make it easier to process

- rich errors are important
  - need more data per error
  - but don’t display it all by default
why should some errors be hidden?

- some "errors" are normal and expected
  - if you ever saw a noisy SMART daemon...
  - or ECC memory has a expected corrected error rate

- let’s call them events
  - they’re not really errors

- hardware errors are often bursty
  - but individual events in a burst not too interesting
  - and on large clusters too much data

- they’re still useful to see trends
  - and should be accounted per component
  - don’t belong in normal kernel logs
error processing

- good error processing needs a lot of state
  - and also policy
  - GUI interfaces for important errors
  - or triggering events
- with triggers when exceeding thresholds
- complex decoding
  - identifying components using firmware help
  - probably not a good idea in the kernel
- one corner case is fatal errors where the kernel has to panic
  - the kernel needs to do limited decoding at least
  - but most errors are not fatal
- need user space for rich error processing
  - we already have it with klogd/syslogd
  - just too dumb
errors vs event tracing

- **normal event tracing aimed at debugging**
  - so higher overhead is ok

- **error handling should be always on**
  - has to work seamlessly in the background

- **small footprint crucial**
  - particularly in memory
  - and in dependencies

- **requirements and tools are quite different**
  - should not be mixed up
  - possibly reuse some infrastructure
  - but only if it has extremely low overhead
so what’s the master plan?

- right now for platform errors (MCE, APEI, PCI-AER)
  - keep basic one line errors in printk with an identifier
    - but only for serious errors or occasionally output for trends
    - strictly rate limited
    - possibly extend KERN_* for severity

- but add structured record on second channel
  - similar to /dev/mcelog, but ascii in sysfs
  - few record types for different types
  - using standard formats (e.g. CPER)
master plan user space

- a standard error daemon
  - light weight to always run
  - has knowledge over basic error types
  - accounts events
  - hooks for automated action
  - simple network protocol interfaces

- extension of mcelog for more errors
  - PCI errors, APEI
  - more in the future?
Questions?
Backup
kernel error problems

- some happen from NMI like contexts

- have to use lockless data structures
  - can cause problems like livelocks

- requires preallocation, potentially wasting a lot of memory