Sensors and Management for Server Appliances

From stock FreeBSD to enterprise ready

Joshua Neal

NetApp, Inc.
495 E. Java Dr
Sunnyvale, CA

joshuan@netapp.com

BSDCan 2009

The appliance market

What is an appliance?

- Generally describes a highly integrated sofware and hardware solution.
- Ranges from low-end embedded devices to high-end server appliances.
- Low-end platforms tend to use highly integrated, custom hardware.
- High-end platforms range from totally commercial-off-the-shelf (COTS) to fully purpose-built hardware.

The high-end hardware may overlap significantly with the server market, leveraging the same technologies.

This presentation focuses on the high-end segment, with a bias toward platforms based on PC architecture.

Out-of-box experience (OOBE)

Enterprise customers buying server appliances want systems that "just work."

The real world conspires against this goal:

- Environmental issues: temperature, power
- Component failures: mechanical, electrical

Appliance must provide:

- Hardware sensors that can detect problems
- Software to perform sensor monitoring
- A sensible policy
- UI functionality to access sensors

COTS hardware is likely to provide the first three, but any platform customization generally requires additional effort to support.

The challenge is growing

Inside the box:

- Increasing system complexity
- Increasing number of sensor interfaces and types
- More field replaceable units (FRUs)
- Pushing the envelope of cooling and power

And outside the box:

- Higher power density
- Trend toward warmer data center temperatures
- Trend away from "clean-room" data centers

Default behaviors

The default behavior may be less than ideal:

- No response to event
- Sudden death
- Dreaded reboot loop
- Logging only
- Custom policy not implemented

Preferred:

- Alerting
- Fault reporting (LED, LCD)
- Identify and perform corrective action
- Clean shutdown and/or graceful failover
- Predicting failures

Beyond simple reporting

What did we need to add?

Required ability to interpret sensor values and respond appropriately.

Required UI for monitoring sensors, adjusting policy (where allowed).

Required unit test hooks, ability to simulate sensors and actions for testing.

Traditionally each sensor interface has its own UI, test hooks, and policy mechanism.

Much duplicated effort, dissimilar levels of functionality.

What if we defined a common abstraction for sensors and then implemented some kind of generic, unified policy engine?

Common abstraction for sensors

What is a sensor?

Easiest to define are traditional physical sensors:

- Correspond directly to some physical measured quantity
- Analog, measuring a range of values (e.g. temperature)
- However, may not always report value (e.g. uninitialized, failed, or busy)

Not necessarily limited to those; other possibilities include:

- Discrete sensors, returning one of a set of values (e.g. present/not present)
- May be measuring a software state or quantity
- May be output of a complex failure function

Sensor metadata

Sensor interface may have associated metadata:

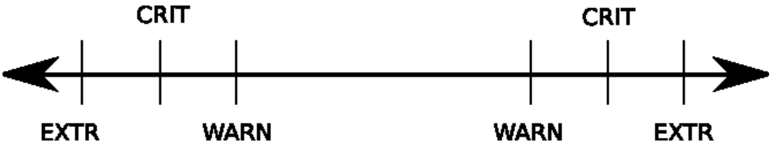
- Sensor name
- Sensor units
- Conversion formula
- Entity/FRU association
- Sensor relationships (e.g. presence, voltage)
- Thresholds

Sometimes discoverable via interface-specific protocol.

If not, platform-specific configuration data is needed.

Sensor thresholds

It is often useful to define a set of ranges that are used to interpret sensor values, commonly defined by a set of thesholds.



Popular thresholds:

- High/low extreme
- High/low critical
- High/low warning

Hysteresis helps prevent oscillations between states.

Sensor access methods

Whether sensors are polled versus interrupt driven may vary by device.

Considerations:

- Sensors are generally not performance critical.
- Sensors are generally not the primary focus of the appliance.
- Polling too frequently may reduce system performance.
- Polling too infrequently may miss important events.
- Interrupts may allow the lowest overhead, while still preventing missed events.

Sensor interfaces: I^{2}C®

I^{2}Cis a two-wire electrical interface standard introduced by Phillips®.

- Most popular sensor interface (vs. SPI, 1-Wire®, etc.)
- Slow speeds (100kHz, 400kHz)
- Low cost to implement, shared two-wire bus and inexpensive devices
- May be driven (bit-banged) by GPIO (with increased CPU overhead)
- Reliability issues (limited error detection, tendency to get stuck)
- Not designed to be discoverable (heuristic probing is sometimes possible)

Sensor interfaces: I^{2}C® protocol

Two data lines, devices either drive bus low or leave alone (external pull-ups).

SCL: clock

SDA: data

Devices take on roles of master/slave. Master always initiates bus tranaction, and typically drives the clock line.

Slave devices are addressed by 7-bit slave address, usually limited to a small set of addresses per device type.

Multiple busses or multiplexers may be needed to avoid address collisions.

Sensor interfaces: SMBus®

SMBus is a subset of I^{2}Cintroduced by Intel®.

Used almost interchangably with I^{2}C® when describing two-wire busses/devices.

Additonal SMBus functionality:

- (Optional) Packet Error Checking (PEC) using CRC-8
- Additional (sharable) interrupt signal line SMBALERT#
- Optional mechanism for auto-discovering devices (ARP) (v2.0)

Sensor interfaces: I^{2}C/SMBus® spotting guide

- Temperature (LM75 and friends)
- Voltage monitor
- Miscellanious GPIO
- Smart batteries
- Register/debug access
- DIMM SPD EEPROMs (incl. DDR3 temp sensors)
- Add-in cards (PCI/PCI-X as well as PCIe)
- Optical transceiver modules (e.g. SFP modules)
- IPMI SSIF interface

Sensor interfaces: IPMI

Intelligent Platform Management Interface (IPMI)

- Standardized interface to a management subsystem (microcontroller)
- Usually a Baseboard Management Controller (BMC) located on or attached to the system board via a proprietary connector
- Most common in server hardware, where it helps to offload responsibilities from main CPU
- May have dedicated or shared console/network interfaces

IPMI 2.0 spec:
http://www.intel.com/design/servers/ipmi/spec.htm

Sensor interfaces: IPMI features

Key IPMI features include:

- Unified protocol for accessing sensors
- Sensor Data Record (SDR) repository for sensor metadata
- SDR includes sensor-to-entity (FRU) mapping
- Proivdes persistent System Event Log (SEL) for event logging
- Well-defined host interfaces (KCS, SMIC, SSIF, BT)
- Watchdog functionality and NMI generation
- Chassis power control
- Provides mechanisms and protocols for remote access to BMC
- Serial-over-LAN (SOL) protocol

Sensor interfaces: ACPI

Advanced Configuration and Power Interface (ACPI)

- Provides interface layer between OS and hardware sensors.
- Complex standard defining virtual machine interpreter for hardware control.
- Defines a common abstraction for common power management and system control functions. Sensors and metadata can be discovered via this API.

Sensor interfaces: ACPI (continued)

The ACPI interface can support a number of sensor types, including:

- Power managment
- Thermal sensors
- Battery monitoring
- Fan control
- Extra buttons
- Ambient light
- Embedded Controller (EC) interface

Most useful for COTS systems, especially laptops that must run off-the-shelf OS.

Appliance vendors may choose to save the cost of implementing ACPI and implement direct control in the application SW instead.

Sensor interfaces: SES

SCSI Enclosure Services (SES)

Attached versus standalone:

- Attached uses functionality built into a SCSI target to tunnel commands to logic on the SCSI backplane
- Standalone provides dedicated target endpoint found on SCSI/SAS backplane, or as part of a SAS expander solution.

Used for management of external shelves; may also be used in appliances with an internal backplane.

Responsibilities may include control of storage status LEDs and fan and power monitoring for backplane, or even entire chassis.

Sensor interfaces: SES model

Each SES enclosure is reported as one or more subenclosures. Each subenclosure may have a set of elements.

The set of predefined element types includes:

- Power supply
- Cooling element (fan)
- Temperature
- Voltage sensor
- Current sensor
- Audible alarm

Sensor interfaces: SES protocol

The SES protocol uses the SCSI SEND DIAGNOSTIC and RECEIVE DIAGNOSTIC RESULTS commands to talk to the enclosure's management processor.

Each of these commands operates on a diagnostic page, which is specified by its 1-byte page code.

Each element type defines its own encoding for element control and element status, containing common fields as well as fields unique to that particular element type.

Sensor interfaces: device-specific

Devices with integrated sensors, accessible via device-specific interface.

Examples:

- CPU/chipset thermal sensors
- Hard drives (via S.M.A.R.T.)
- Add-in cards
- USB peripherals

Each device tends to require its own specialized access method for querying sensor values and metdata (if any).

Existing applications

I^2C/SMBus®

- iic(4), smbus(4)
- smbmsg
- mbmon, healthd
- bsdhwmon (cancelled)

IPMI

- ipmi(4) (KCS, SMIC, SSIF)
- ipmitool
- freeipmi

Existing applications (continued)

SES

- ses(4)
- /usr/share/examples/ses

ACPI

- acpi(4)
- sysctl (e.g. hw.acpi.thermal.*)

Device-specific

- CPU: coretemp(4), amdtemp(4) - sysctl dev.cpu.*.temperature
- HDD: smartmontools (smartctl -A)

A new, unified sensor architecture

Goal: Implement a reusable policy engine for multiple hardware platforms.

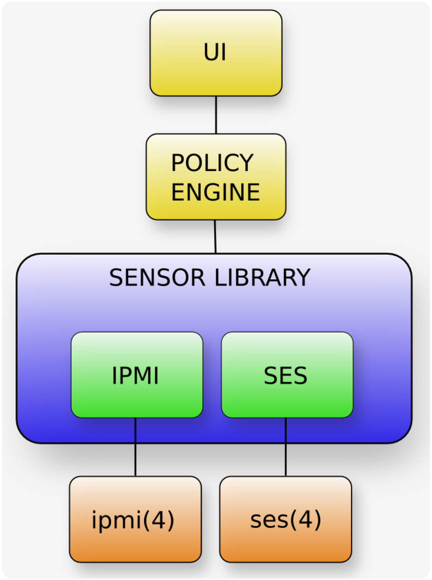
Some sensor abstraction was needed to support various types of sensors.

Significant effort has gone into implementing a unified kernel sensor framework for FreeBSD (e.g. Constantine A. Murenin's port of OpenBSD sensor framework).

However, for the most complex protcols, IPMI and SES, most of the protocol can (and should be) implemented in userspace. The kernel component can be limited to a lightweight, generic transport with only a few supported operations.

An application library is a better place to provide a unified sensor interface.

A new, unified sensor architecture



kqueue

Needed to provide an event delivery mechanism to userspace.

Kqueue seems to fit the bill nicely:

- The application can wait on both sensor events and socket (command interface) events using a single API.
- Kqueue allows passing additional information along with event back to the userspace, may be used to avoid additional syscall.

Implemented kqueue handlers for sensor devices that need to send events to the userspace.

kqueue(2), sys/event.h

```
struct kevent {
    uintptr_t       ident;    /* identifier for this event */
    short           filter;   /* filter for event */
    u_short         flags;
    u_int           fflags;
    intptr_t        data;
    void            *udata;   /* opaque user data identifier */
};

#define EV_ADD      0x0001    /* add event to kq */
#define EV_DELETE   0x0002    /* delete event from kq */
#define EV_ENABLE   0x0004    /* enable event */
#define EV_DISABLE  0x0008    /* disable event (not reported) */

#define EV_ONESHOT  0x0010    /* only report one occurrence */
#define EV_CLEAR    0x0020    /* clear ev state after reporting
```

kqueue(2) (continued)

```
int
kqueue(void);

int
kevent(int kq,
    const struct kevent *changelist, int nchanges,
    struct kevent *eventlist, int nevents,
    const struct timespec *timeout);
```

kqueue limitations

- Only one integer value passed back to userspace.
- Not really a queue, multiple events coalesced into one event.
- Preferrable to use level-triggered mechanism.

Idea: Implement a sensor event queue in each driver, which tracks events and any device-specific event information

kqueue mechanism used to report level-triggered "sensor event queue contains events"

Sensor event queue read via ioctl()

Future directions

Since kqueue is designed to handle many thousand connections efficiently, event per sensor (or multiple per sensor) may be reasonable.

Add kernel layer to attach drivers to, to minimize effort required to export sensors.

Better support for hotplug notifications.

Something completely different?