OpenAFS Instrumentation Framework

Tom Keiser
Sine Nomine Associates

Introduction

- Project was commissioned by a client of Sine Nomine
- Primary goal is to provide a universal instrumentation framework to monitor and identify performance bottlenecks
- Development, testing, and platform porting efforts are ongoing
- Current patch against OpenAFS HEAD touches $1,100$ files, and is $127,000$ lines, and growing...
- An alpha patch was committed as the `instrumentation` branch of OpenAFS CVS in 01/2007

Motivating Problems

- Current afs monitoring techniques are reactive
- Continuous polling is required to identify system faults
- Knowledge of many disparate monitoring and debugging technologies is required
- Proactive monitoring is needed to:
  - find and isolate faults more quickly
  - gather a historical record of the conditions leading up to faults
  - allow for future fault data mining

Motivating Problems Part II

- Current monitoring and debugging systems have races
  - fetching state in N unit blocks inherently racy
  - fetching many units of state as a single transaction is unfair and introduces jitter into production workflows
  - snapshotting full system state is unacceptable due to the required serialization and jitter
- Proactive monitoring allows us to avoid bulk state fetches by only sending state:
  - when interesting events happen
  - which is interesting to the user

A Taxonomy of Instrumentation Techniques

- There are a number of important properties to consider when designing an instrumentation system:

  solicitation must telemetry be requested, or will it be provided without solicitation?

  synchronousness poll versus publish/subscribe semantics

  stability is the telemetry data cacheable, or ephemeral? put another way: does temporal correlation between data points from the same source imply value correlation?

Taxonomy Part II

- AFS exhibits four major classes of instrumentation:

  events delivery of ad-hoc state (e.g. `fstrace`)

  queries highly-structured telemetry acquisition methods (e.g. `rxdebug`, `cmdebug`, ...)

  static statistics statistics which are statically allocated (e.g. `xstat`, `rx_stats`, ...)

  dynamic statistics statistics related to dynamic objects (e.g. `rx_conn`, `rx_call`, and `rx_peer` statistics, ...)

- unfortunately, there is no common framework — each subsystem reinvents the wheel

Taxonomy Part III

- Now, we can classify each of our instrumentation classes using the previously discussed properties:

  events unsolicited, asynchronous, unstable

  queries solicited, synchronous, both

  static statistics both, both, both

  dynamic statistics both, both, unstable

- bottom line: statistics are hard to classify — we can treat stats updates as trace events, or we can poll for current stats values on an interval

Design Goals

- Support all four instrumentation classes from the taxonomy
- Minimal disabled probe overhead
- Integrate seamlessly with enterprise monitoring tools
- Provide linear MP scaling
- Write code using extensible and pluggable APIs
- Provide for distributed telemetry processing and distributed transaction correlation
- Provide a scripting language interface to lower the barrier to entry
- Combine all instrumentation into a single namespace

Existing Technologies

- The chosen design borrows elements from a number of contemporary technologies:

SNMP hierarchical namespace, agent/console architecture, trap/get methods

Sun DTrace generator/consumer model, dynamic probe registration, complex probe actions, and data postprocessing/aggregation

Solaris kstat abstraction for managing and updating statistics

z/VM per-cpu ring buffers of fixed-length trace records

AIX Trace Framework excellent example of a developer-oriented tracing framework

Prerequisites

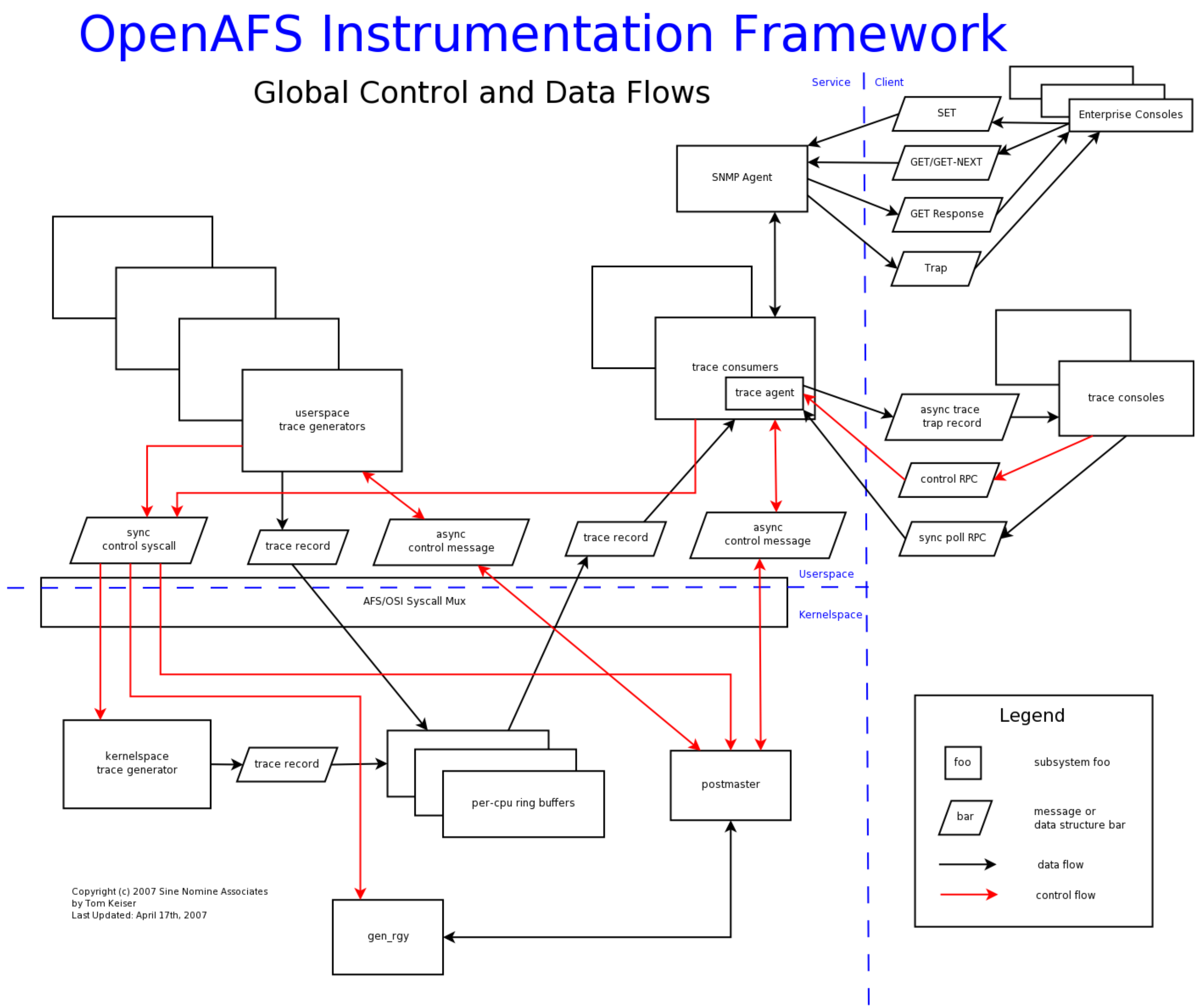
- Before serious work on instrumentation could begin, we needed a robust, portable runtime abstraction
- Historically, DCE/DFS, and to a significantly lesser extent, AFS, have had runtime abstractions called osi:
  - interfaces were sprinkled throughout the code with seemingly little order
  - no naming consistency
  - primarily aimed at kernel code
  - typically valued portability over performance
- From these deficiencies came the birth of libosi (the Operating System Interface library)

Implementation

What is libosi?

- libosi provides functionality similar to APR and NSPR
- Frequently, code written to libosi interfaces can be recompiled for userspace and kernelspace without any preprocessor ifdef's
- Provides numerous high-performance interfaces, such as:
  - atomic operations
  - per-cpu memory
  - numa-aware memory pools
  - high-performance statistics abstractions
  - and quite a bit more!

Architecture



Probe Naming

- Namespace closely resembles SNMP
- probe names are arranged as a dot-delimited hierarchy
- at present, there are 983 probes in the tree
- here are some examples:
  - `rpc.rx.conn.new`
  - `rpc.rx.srv.callQ.enqueue`
  - `vol.volume.actions.attach.begin`
  - `legacy.icl.afs_trace.CM_TRACE_STOREPROC`
  - `db.ubik.client.events.mark_server_down`
  - `srv.fs.file.CopyOnWrite`

Producer/Consumer Model

- We utilize an M:N producer/consumer model
- Instrumented processes, and instrumented kernel components, can emit trace data from activated probe points
- Emitted trace data is pulled out of kernel ring buffers by consumer processes
- Consumers make trace data available to:
  - local code written to the consumer C apis
  - local user-defined data postprocessing and analysis routines
  - remote trace consoles via Rx
  - SNMP agents

Producer/Consumer Model Part II

- Consumer processes perform the following chain of operations on incoming trace data:
  - Trace records are queued for probe id to probe name resolution
  - Local interested parties are resolved, and probe data forwarded
  - Remote consoles with interest are identified
  - Probe data is encoded for remote transport
  - Traps are sent to the appropriate remote consoles

Remote Tracing

- We deliver telemetry via Rx, and SNMP
- Remote tracing is necessary in order to correlate events in distributed transactions
- For the sake of flexibility, we support both polling and asynchronous trap methodologies
- Asynchronous traps operate via a priori registration of probe filter expressions, e.g:
  - `*`
  - `srv.fs.rpc.*`
- Using this framework, it is possible to understand performance bottlenecks in large distributed systems

Data Postprocessing

- In order to cut down on bandwidth, the system incorporates the ability to perform data postprocessing and aggregation before transmitttal across the network
- This subsystem is referred to as the analyzer library
- Analyzer operates in a manner similar to a digital logic simulator
- By composing graphs of these components it is possible to develop complex analysis routines
- Individual outputs from the analysis library may be subscribed to, and the results may be transmitted to remote consoles — all in the same manner as normal probe data

Data Postprocessing Part II

- individual analysis components perform simple tasks such as:
  - integer arithmetic
  - boolean logic
  - timing
  - counting
  - summation
  - memory functions
  - etc.

SNMP

- SNMP is a core requirement for integrating with enterprise monitoring systems
- development effort is still underway

The SNA Instrumentation Team

- Dr. David Boyes — SNMP Framework Architect
- Derrick Brashear — Random help (when he has time)
- Tom Keiser — Lead Programmer
- Evan Macbeth — Project Manager
- Mike Meffie — C Unit Testing, Debugging, Releases, etc.
- Adam Thornton — Perl Unit Testing and SNMP Development

Questions?