SAND2020-7990C

Sandia National Laboratories

Supporting Dynamic Event Monitoring in the Lightweight Distributed Metric Service (LDMS)

aka: LDMS_Streams

OGC

Open Grid Computing, Austin, TX

LDMSCON2020

Tom Tucker, Ann Gentile, Jim Brandt

U.S. DEPARTMENT OF ENERGY NNSA

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Introduction

LDMS is designed to minimize the impact of system and application related data collection on running application. Its primary model of operation is pull-based to minimize the functional demands and hence impact on the compute nodes.

However, for dynamic and/or irregular events, both log and numeric, the natural model would be push-based.

A combination of models would enable low-overhead, low-latency handling of both regular system monitoring data and event data. The ultimate goal is to enable the examination of application performance data in conjunction with system conditions

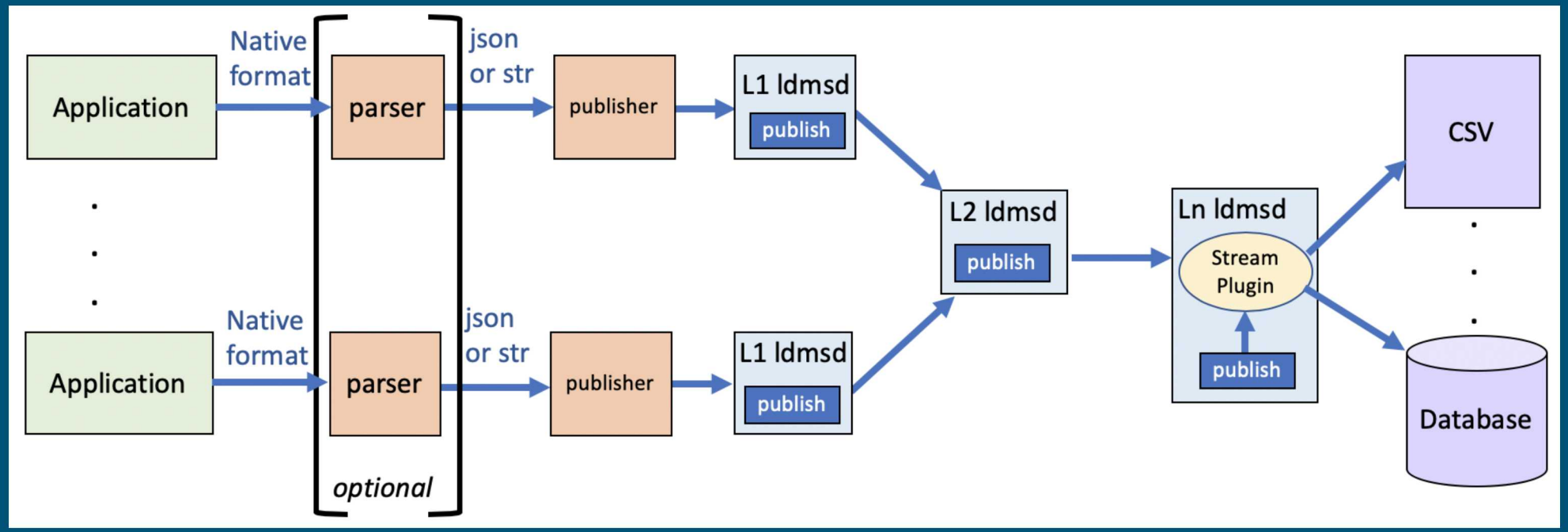
Implementing both within one architecture enables (a) ease of use and (b) use of the existing LDMS transports (Socket, uGNI, IB Verbs) and authentication without additional development required.

We describe the LDMS_Streams functionality and API to enable arbitrary string-based push events and provide use-cases demonstrating the utility of combined push and pull based data flows.

LDMS Architecture: Background

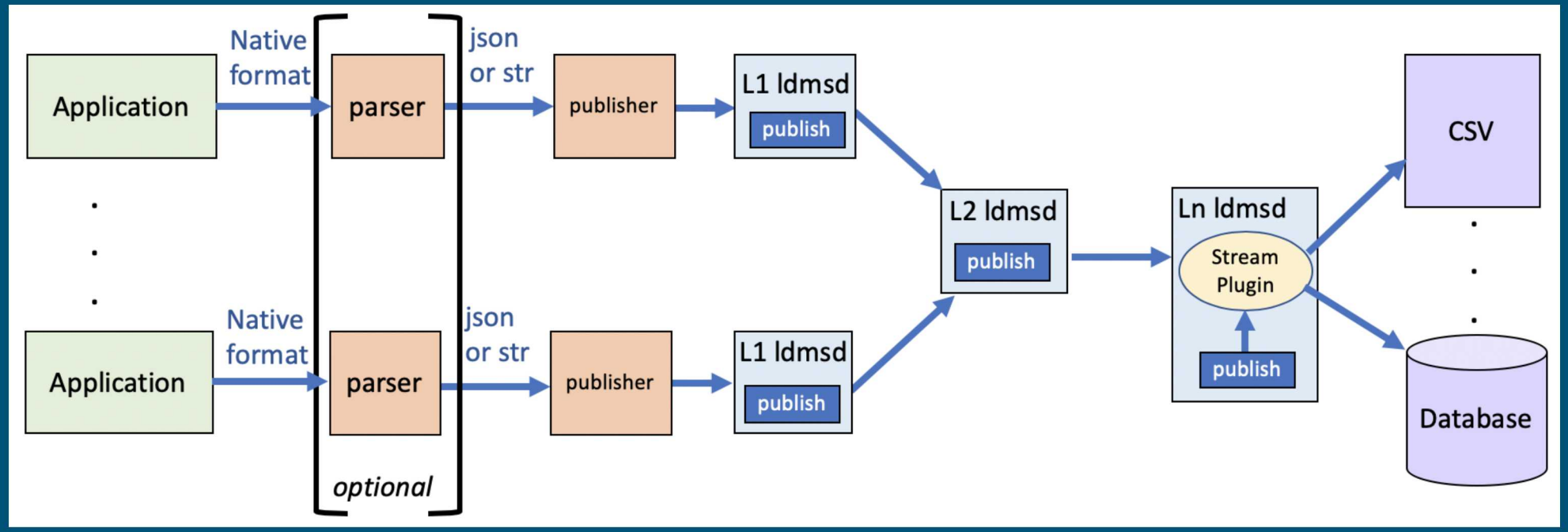
- LDMS is plugin-based
  - Sampler plugins – data is stored in a well-defined construct called a metric set
  - Store plugins
- Aggregator daemons are typically configured to pull the data from producer ldmsd
- Minimize impact:
  - Data collected by sampler plugins is stored in well-defined memory space and overwritten by each new sample
  - Metric set: Data is split into meta-data and data values. Only the latter is transported each time
  - Pull mode minimizes the CPU and memory requirements of the compute node ldmsd
- Aggregator Daemons with *store plugins* call the *store* function when they pull data relevant to that plugin

Push-based Data Flow: LDMS Streams



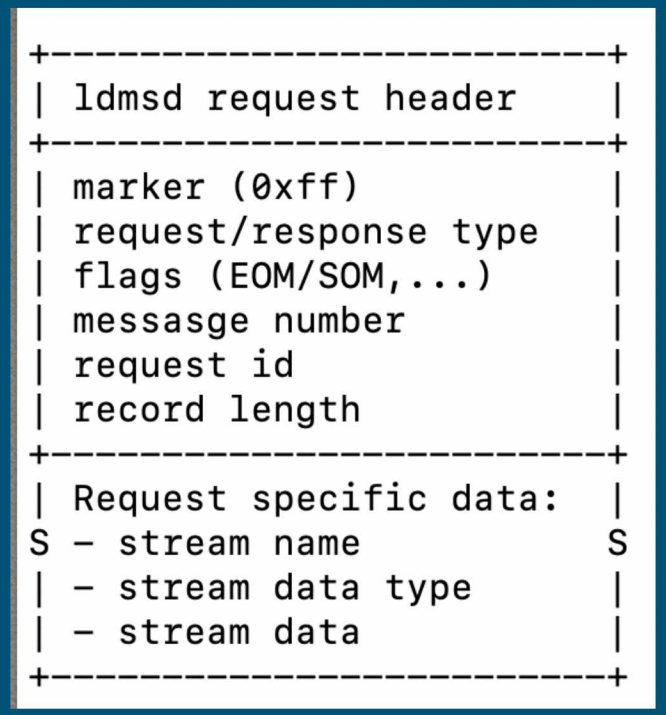
- Data is published to a running ldmsd
- Syntax includes a *tag* that uniquely identifies a stream. Multiple messages can be associated with a stream
- Remote subscribers: ldmsd can subscribe to producer ldmsd and particular streams to be pushed that data
- Local Subscribers: Plugins can also subscribe to a stream. Subscriber callback function is invoked by the ldmsd whenever a message for that stream is pushed to the ldmsd

Push-based Data Flow: LDMS Streams (cont'd)



- A plugin can write out streams data to a store just like a canonical store plugin writes out pull-based data
- Stream data is more flexible in its type: json or str typed messages as opposed to metric set
- May require case-specific unpacking at the aggregator/store, but that is off compute node
- May require case-specific adjustments at the on-node publish (e.g.,json translation), however we seek to minimize the overhead and intrusion at the on-node publish

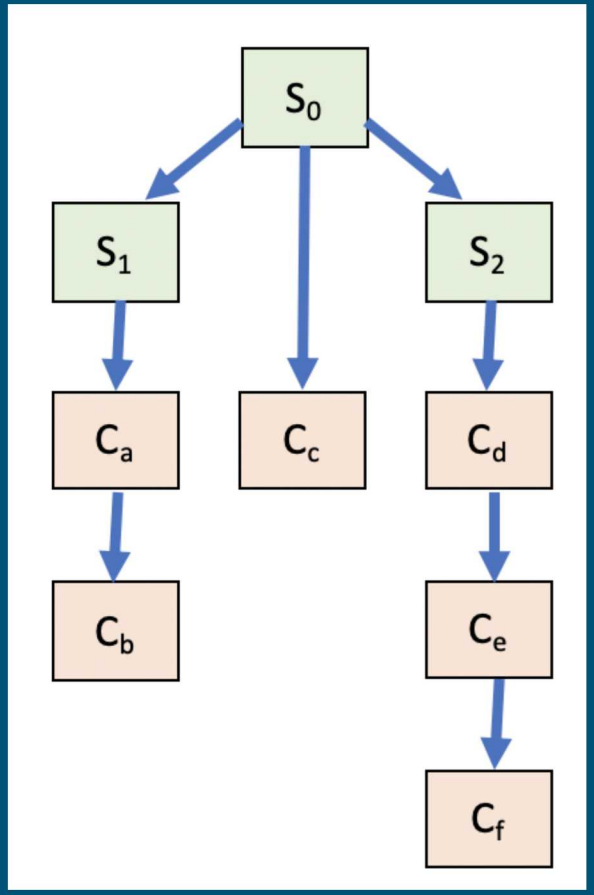
Design and Implementation – Message Internals



- *marker* - begins every message and ensures that the receiver can reliably determine that the data is the start of a new message
- *request/response type* - specifies the type of the configuration request. Streams data has two types:
  - LDMSD_STREAM_SUBSCRIBE_REQ
  - LDMSD_STREAM_PUBLISH_REQ
- *message number and request_id*: handle requests larger than the transports' maximum record length
- Data particular to the request:
- LDMSD_STREAM_SUBSCRIBE_REQ – *stream name*.
- LDMSD_STREAM_PUBLISH_REQ -- *stream name, stream data type*, and the *data* being published to that stream.
  - Data type is one of LDMSD_STREAM_STRING or LDMSD_STREAM_JSON.

Design and Implementation - Delivery

- LDMSD_STREAM_SUBSCRIBE_REQ :
  - Remote – another ldmsd wanting to be pushed a stream
  - Local – a plugin wanting the callback function to be invoked for a streams data
- A stream client is created in the ldmsd. Stream clients are stored in an RB tree indexed by the stream name.
- LDMSD_STREAM_PUBLISH_REQ:
  - JSON data is validated and put into a structure for easier parsing by the callback function.
  - String data is *not* validated
  - RB Tree is walked to determine clients for that stream



- Delivery is best effort – no reconnect/resend
- Data is not cached – subscriber only receives data published after subscription

Design and Implementation: Function Calls

- Local Subscribe:

```
extern ldmsd_stream_client_t ldmsd_stream_subscribe(const char
*stream_name,  ldmsd_stream_recv_cb_t cb_fn, void *ctxt);
```

- Callback typedef:

```
typedef int (*ldmsd_stream_recv_cb_t)(ldmsd_stream_client_t c, void
*ctxt,  ldmsd_stream_type_t stream_type,  const char *data, size_t data_len,
json_entity_t entity)
```

- Publish:

```
extern int ldmsd_stream_publish(ldms_t xprt, const char *stream_name,
ldmsd_stream_type_t stream_type, const char *data, size_t data_len);
```

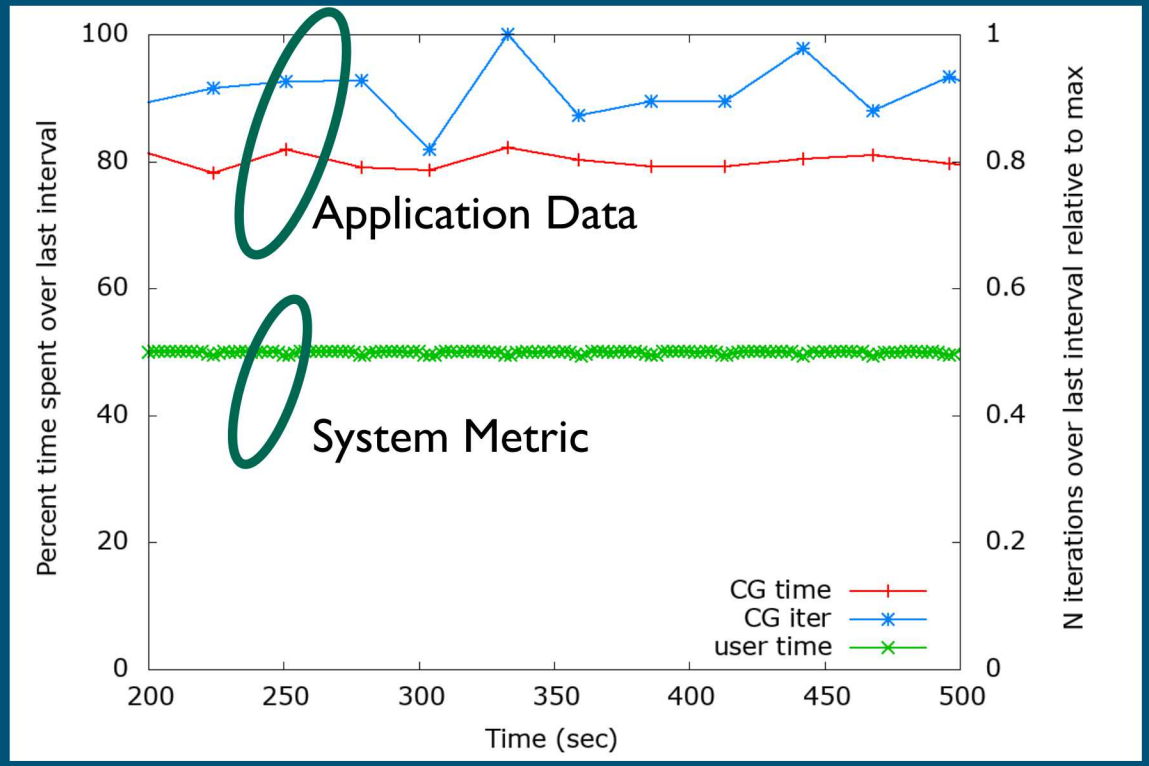
Examples in the hello_stream sampler and store directories in the V4 release

Use Case 1: Combined Application and System Data

- MILC – NERSC_TIME instrumentation: the time spent in particular algorithm sections - conjugate gradient, calculation of the fermion force, and time spent in restoring the fermion links.
- Augmented code to include timestamp and a identification tag
- LDMS setup is the canonical set up with the additional stream subscription config lines
- Application launch line includes directing the output to a parser that reformats tagged lines as json and publishes them to a stream
- Canonical store plugin stores metric data; stream subscriber plugin stores stream data

Example: Single node MILC run. Conjugate Gradient timing and iteration information output by the code and published. CPU information collected via LDMS via canonical methods.

- CG output at variable ~30 sec intervals determined by the code
- LDMS system data collection at 2 sec intervals

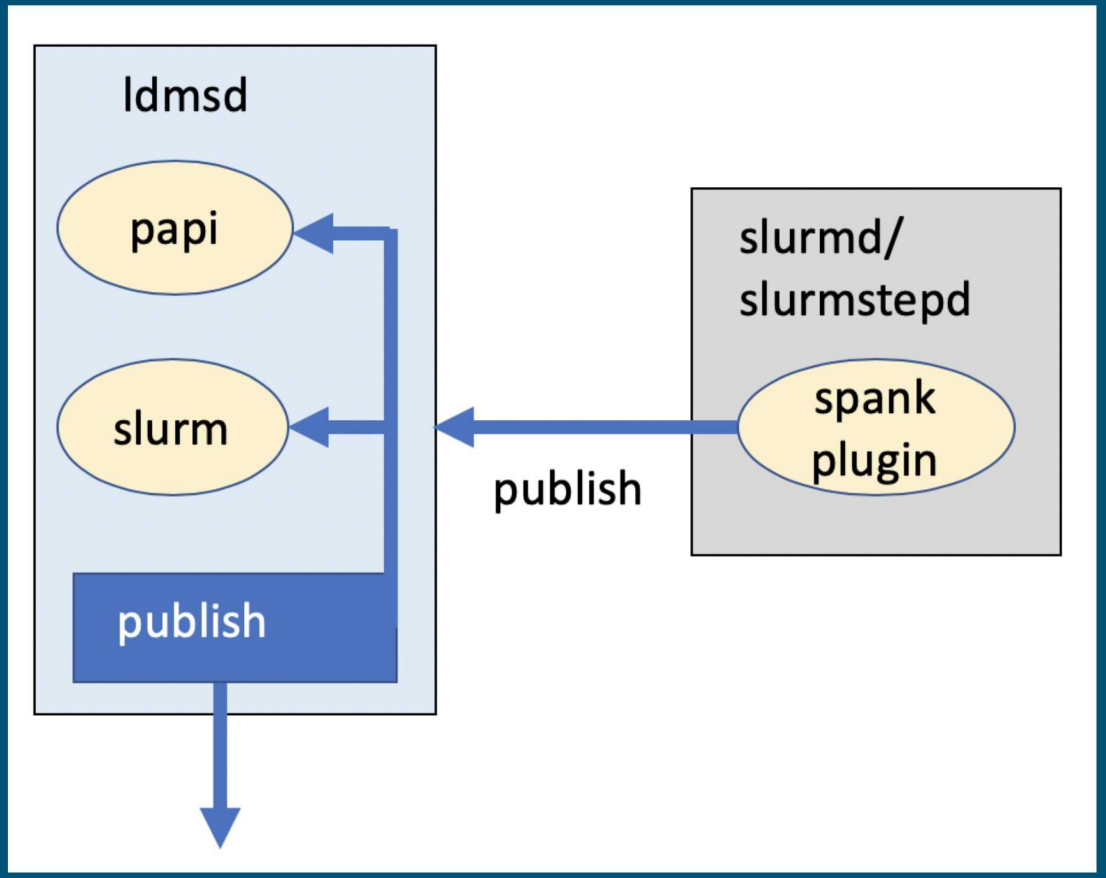


Use Case 2: Combined Log File and System Data

- Log data is irregular, dynamic data
  - syslog - can be written to a file or redirected to a port
  - application log data
  - tracking a file using inotify or listening on a port can publish the live data in string format to the ldmsd_stream interface.
  - At the store, contextual extraction may be required to enable actionable response or plotting.

Use Case 3: Slurm Plugin for Job Information

- Slurm loads the LDMS libslurm_notifier plugin on each compute node.
- Each time a job is changes state (starts, terminates, pauses), this slurm plugin is used to get job information such as identifier, state, size, uid, gid, time, etc.
- The stream publish interface is used to get relevant information to the ldmsd.
- Through subscription to the "Slurm" stream other sampler plugins can gain access to this information and include it in their (pull-based) metric sets.



Upcoming Work

- Working with Trilinos developers to inject application progress data, based on Teuchos Timers, into the LDMS stream (ASC FY21 L2 Milestone)
- Determining visual representations and analytics for the combined data
- Application Teams seeking to inject science variables at run time
- Supporting plugin development:
  - refactoring so that a Streams' Plugin callback and a canonical Plugin's store function can share the same code