StreamFS: Streaming file data abstraction for sensor deployment data collection and analysis

1 Contributions

1. Stream file abstraction.

2 Evaluation

1. Compare with simple logging.
2. compare with scribe.
3. comapre with chukwa.

3 Introduction

Machine generated data is increasing with the rise of cloud-computing and the increasing number of network-enabled embedded sensing.

Log (and data logging) definition:

1. A record of computer activity used for statistical purposes as well as backup and recovery.
2. The process of using a computer to collect data through sensors, analyze the data and save and output the results of the collection and analysis.
   - Data logging is commonly used in scientific experiments and in monitoring systems where there is the need to collect information faster than a human can possibly collect the information and in cases where accuracy is essential. Examples of the types of information a data logging system can collect include temperatures, sound frequencies, vibrations, times, light intensities, electrical currents, pressure and changes in states of matter.
3. is the process of recording events, with an automated computer program, in a certain scope in order to provide an audit trail that can be used to understand the activity of the system and to diagnose problems.
   - Logs are essential to understand the activities of complex systems particularly in the case of applications with little user interaction (such as server applications).
   - It can also be useful to combine log file entries from multiple sources. This approach, in combination with statistical analysis, may yield correlations between seemingly unrelated events on different servers. Other solutions employ network-wide querying and reporting.

Most system focus only on log collection. StreamFS provides facilities for log collection *and* analysis.

Transducers are defined for any file. We use the filesystem to collect the logs. We overload the pipe abstraction for dataflow programming.

Most analysis systems are focused only on data collection. We provide a framework for collection, sharing, and processing of log data, whether it comes from racks in a data center or sensors in a building. We provide a structured format for this log data and impose a transformation function on the data to give it a timeseries data structure. By doing so we can efficiently store, process, and extract the data for processing.

Our contributions:

1. We introduction a new kind of file called a *stream file* for logging and querying data and define the read, write, and execution semantics.
2. We re-introduce the notion of transducers to transform semi-structured into timeseries-structured data and back.
3. We design and implement a web-service called StreamFS that provides multiple interfaces, including a mounted POSIX compliant filesystem mount, which implements our stream file abstraction.

4 Design goals and rational

1. The data pipeline between sensor deployments and logging systems and application is much too complicated.
2. Most deployments continue to go through the same steps when writing their data-logging infrastructure:
   (a) Write code to receive data from sensors.
   (b) Format data into log format and write it to a file or set of files.
   (c) Write code to parse log files and import into analysis application.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

3. Real-time analysis is a separate, but similar process:
   (a) Write code to receive data from sensors.
   (b) Pass data to code that consumes one or more data points and outputs a results or trigger.
   (c) Write output to file for post-analysis of events.
   (d) Write code to parse event-log files and import into analysis application.

All deployments essentially write they own data collection code, data translation and import code, and data extraction and application-specific formatting code. This is redudant and wasteful of the programmer's time and increases the full analysis phase, since for each new deployment or experiment this process is repeated.

4.1 Data in, data out

Data from deployments can be divided into three classes: *temporal, descriptive, and contextual*. In order to determine the kind of data that is being collected we must label it with descriptive information that includes the units of measurement, sensor make and model, the owner of the sensor, and calibration parameters. Since sensors are typically embedded in the physical environment, it's important to capture the contextual information about their placement. For example, sensors placed throughout a building are tagged with room or system-placement information (such as the heat pump or air conditioner). Finally, when taking physical measurements there is a temporal component to the data to capture the time-varying nature of the phenomena being measured.

Most data-collection system designers either combine all three classes into a single data store or separate out the temporal data from the other two. When all the sensors are deployed by the same organization and follow the same conventions, this is okay. However, it becomes increasingly difficult to organize deployment information across systems designed by different vendors and that complexity is reflected in the code; as developers write different drivers to translate between formats into a common schema and data store. Similarly, on the application side, different applications expect different incoming data formats and application developers write application-specific formatters that translate the data from the stored format to the application-specifc format.

4.2 Security

Access to sensor data across different systems also adds to code complexity, as the developer deals with security policies and mechanisms across systems. Developers need to reason about both access to the data and access to the sensor or actuator. In some cases security policy is context-driven, whereby clients are granted access based on the location of the request or the location of the sensors. For example, you may only want to grant global access to sensors in shared spaces, and limit access to personal sensors. In other cases, security policy is granularity dependent. For example, you may want to grant access to aggregate power-data for an entire building, but not to individual raw data streams.

4.3 Analysis, processing, and sharing

A fundamental problem in dealing with sensor data is that it typically needs to be cleaned. In many cases, this can be easily accomplished with thresholding, interpolation, and duplicate removal. Developers typically leave this concern to external applications. However, since dirty sensor data is fundamental, we argue a cleaning facility should be part of the data collection system itself, rather than be left to external applications.

4.4 Common interface

Today, a broad range of applications are built on top of sensor data. Beyond typical data collection and analysis, new applications are providing real-time analytics and control of sensor-rich environments. Many applications run on single or multi-host environment and others are made to run on mobile phones. Designing the right interface for such a broad range fo applications is a challenge that is typically offloaded to the application developer. We argue that this is indeed the correct choice, however a common organizational principal should be followed across different interfaces for all applications. This makes it easier for the application developer to reason about the the aforementioned issues.

4.5 sMAP, FIAP, CoAP comparison

4.6 Why a file?

All real system implement logging and industry-standard logging system use *syslog* and *NFS*.

5 References