Processing of Flow Accounting Data in Java: Framework Design and Performance Evaluation

Jochen Kögel, Sebastian Scholz

Processing of Flow Accounting Data in Java: Framework Design and Performance Evaluation

Jochen Kögel and Sebastian Scholz

Institute of Communication Networks and Computer Engineering (IKR)
University of Stuttgart
Pfaffenwaldring 47
70569 Stuttgart
{jochen.koegel, sscholz}@ikr.uni-stuttgart.de

Abstract Flow Accounting is a passive monitoring mechanism implemented in routers that gives insight into traffic behavior and network characteristics. However, processing of Flow Accounting data is a challenging task, especially in large networks where the rate of Flow Records received at the collector can be very high. We developed a framework for processing of Flow Accounting data in Java. It provides processing blocks for aggregation, sorting, statistics, correlation, and other tasks. Besides reading data from files for offline analysis, it can also directly process data received from the network. In terms of multithreading and data handling, the framework is highly configurable, which allows performance tuning depending on the given task. For setting these parameters there are several trade-offs concerning memory consumption and processing overhead. In this paper, we present the framework design, study these trade-offs based on a reference scenario and examine characteristics caused by garbage collection.

1 Introduction

Monitoring network characteristics and traffic is vital for every network operator. This monitoring information serves as input for adjusting configurations, upgrade planning as well as for detecting and analyzing problems.

Besides active measurements and passive capturing of packet traces, Flow Accounting is attractive because it is a passive monitoring approach, where information on flows is created in routers and exported as Flow Records using a protocol like Cisco NetFlow [1] or IPFIX [2]. Due to monitoring on the flow level, Flow Accounting provides a good trade-off between the information monitored and the amount of data to store and process. Flow Accounting is mainly used for reporting and accounting tasks, but can also provide input for anomaly detection or extraction of network characteristics [3].

Processing of Flow Accounting data is challenging, since in large networks routers export several hundred million Flow Records per hour. Three common approaches can be distinguished to handle and process this data. First, Flow Records can be stored in a central or distributed database to create reports and

offline analysis at a later point in time. Here, the attributes of Flow Records are often reduced in order to save memory and processing effort. Second, Flow Records can be dumped directly into files for offline analysis without a database. Third, Flow Records can be analyzed online directly in memory without storing them.

We developed a flow processing framework in Java that can read Flow Records from files as well as from the network. Thus, it is suitable for offline and online analysis. The framework processes Flow Records in a streaming fashion, i.e. data flows through a chain of processing blocks. Each block can keep data as long as required for its task (typically using a sliding window) before it forwards the processing results. For tasks like joining and sorting Flow Records, the window size parameter for achieving good processing results depends on characteristics of the Flow Accounting data, which in turn depends on router configuration and traffic characteristics. Hence, these factors influence how long data is kept in memory and thus the overall memory consumption. However, the latter cannot be evaluated as independent metric in Java: more memory consumption results in more garbage collector overhead and thus eventually in reduced throughput.

This paper studies these dependencies. Besides, the processing blocks can be assigned to threads in different schemes, allowing to exploit modern multicore computers. The investigation of these threading schemes is also part of this work.

This paper is structured as follows: Section 2 introduces Flow Accounting, Section 3 presents the design of the flow processing framework, Section 4 shows the result of the performance evaluation, and Section 5 concludes the document.

2 Flow Accounting

2.1 Mechanism and Protocols

Flow Accounting is a mechanism present in most professional routers that keeps counters on per-flow basis. The router exports this information as Flow Records and sends them a collector, where the information is processed further. Flows are identified by a key, which is typically the five tuple consisting of source and destination address, source and destination port, as well as transport protocol number. Routers keep a table (*flow cache*) where data on each flow is stored. The router updates the flow information (e.g. byte and packet count) either for each packet (unsampled) or for a fraction of packets (sampled). Among other information, Flow Records contain the five tuple, the counters as well as start and end time in milliseconds.

Several data formats for sending multiple Flow Records in a packet to the collector exist. Mostly Cisco system's NetFlow format is used. The current NetFlow version 9 [1] and its successor IPFIX [2] are flexible formats based on templates, while the fixed format of version 5 dominates current deployments. Since the data is sent via UDP, packets transporting Flow Records might be lost.

For determining the export time t_x of a Flow Record, there are several criteria in Cisco routers. *Inactive timeout*: if for a flow there is no more packet seen for

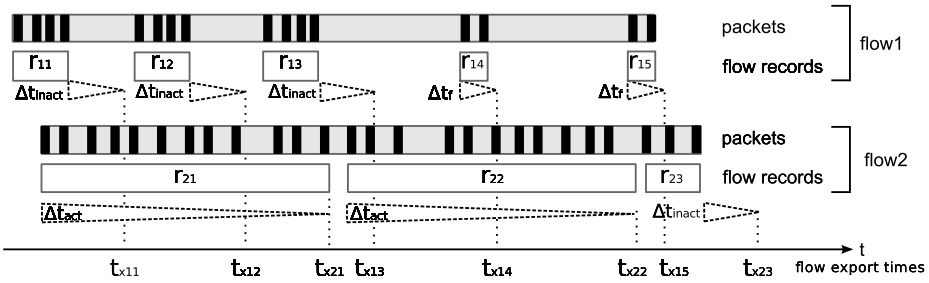

Figure 1. Flow export: timers of different strategies and resulting record order

Δt_{inact}, the flow is exported. *Active timeout*: if a flow has been active for a time period greater than Δt_{act}, the flow is exported. *Fast timeout*: if after Δt_f a flow contains less than n_{fast} packets, the flow is exported. *Cache clearing*: if the cache runs full, the router exports flows earlier than defined by the timeouts.

Some routers determine flow ends also by tracking TCP connection state. The timeouts are illustrated in Fig. 1 for two flows. We can see that information on flows can be distributed across different records with long breaks in between and that records arrive at the collector neither sorted by start nor by end time. These properties have to be considered for processing algorithms that work on a limited window of Flow Accounting data.

2.2 Existing processing tools

In large networks several hundred million Flow Records arrive at the collector per hour. Processing or storing this amount of data is challenging. There are several commercial tools that collect and analyze Flow Accounting data, such as IsarFlow, Arbor Peakflow, Lancope StealthWatch or Cisco Multi NetFlow Collector. These tools follow the common approach of using a database (DB) for offline analysis. In order to cope with the data rate, load balancers distribute the traffic across several collectors that form a distributed DB. These tools often aggregate data as soon as possible (e.g. on time intervals) to reduce storage requirements. Thus, evaluations that need fine grained timing information are not possible. There are also free tools for collection and basic processing of Flow Accounting data, such as *flowtools* and *nfdump*. Free reporting and query tools are *nfsen* and *SiLK*.

Evaluation algorithms that rely on fine grained information are typically processing and memory intensive, thus stream-based approaches have advantages over DB based tools. Processing network monitoring data in a streaming fashion is close to the domain of data stream management systems (DSMS) or complex event processing. Related tools are Gigascope [4] for packet trace processing, the TelgraphCQ DSMS [5] or the network monitoring specific CoMo project [6]. Other tools focus on the dispatching of received NetFlow data [7] or on Flow Query Languages [8]. The presented tools are often limited to a certain domain

and not suitable for extracting and correlating network characteristics or data from different data sources. A data processing pipeline in Java is presented in [9]. Its focus is on processing large objects (e.g. images), thus it is unsuitable for Flow Accounting data.

3 Data Processing Framework

3.1 Architecture

We developed a framework for processing of Flow Accounting data, especially for fine grained analysis of Flow Records e.g. for extraction and correlation of network metrics. This includes tasks like matching records from different sources, calculation of derived metrics, correlation and statistics. In order to handle the huge amount of data, we use a stream-based approach that capitalizes on modern multicore architectures. We decided to take Java as programming language due to the garbage collection feature of the Java Virtual Machine (JVM), which simplifies modular design, and its built-in utility classes for concurrent programming.

Our framework is based on interconnected processing blocks that form a data processing chain. Processing blocks exchange messages containing references to objects (Fig. 2). Each block has at least to implement a data source or a data sink interface, while combinations with several interfaces are also possible (e.g. several inputs/outputs). A processing block is derived from a generic block according to its role. Fig. 2 shows an exemplary chain with two *SourceBlocks* reading NetFlow data from files or the network and forwarding the data to *SinkSourceBlocks* that perform data aggregation to *JoinedFlows*. The two pipelines are merged in a *CorrelationBlock*, which creates correlation result objects that are statistically evaluated in two *SinkBlocks*.

Processing chains are constrained to a directed acyclic graph. While cycles could make sense e.g. for feedback loops to compensate time offsets in the data, this is currently not supported. We designed the framework in a way that threads can be assigned to one or several processing blocks. If two processing blocks run

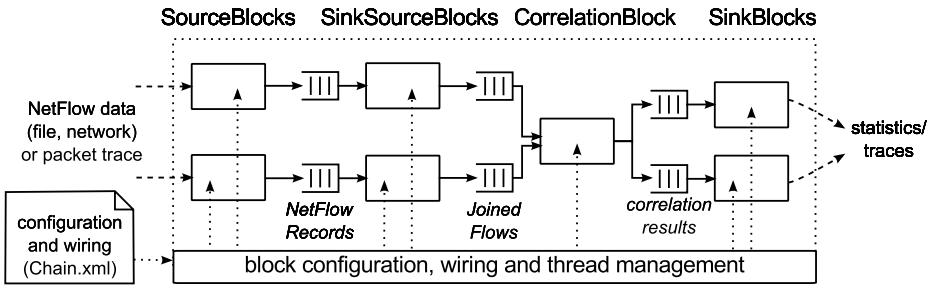

Figure 2. Exemplary processing chain showing basic types of processing blocks

as different threads, they are connected via blocking FIFO queues, as shown in Fig. 2 for the configuration with the maximum number of threads.

At each source interface, an arbitrary number of processing blocks can connect, such that all of them will get references to the objects passed on and can process them independently of each other. We avoid race conditions possibly caused due to concurrent access by not modifying objects after they left the block where they have been created. Due to garbage collection provided by the JVM, no mechanisms to manage the references to objects and freeing memory in case of dropped objects is necessary. This enables a clean design of independent processing blocks.

At startup, the chain is set up by a central component that also performs thread management. Configuration is based on an XML file that describes the chain structure and processing block parameters. For this, we build on the dependency injection mechanisms provided by the Spring Framework [10]. After all objects are created and wired as defined, threads are started and the *SourceBlocks* starts delivering data to the chain. Shutdown is initiated by a shutdown message in downstream direction, e.g. if readers run out of data. If the chain contains parallel paths that are merged in a correlator, this mechanism is not sufficient for proper shutdown of upstream blocks that still have data. In such cases, upstream shutdown notification is performed by deregistering connections from upstream blocks, which will then shut down.

3.2 Processing blocks

In terms of processing tasks there are two basic classes of processing blocks: *window-based blocks* that keep data over a sliding window, and *window-less* blocks that perform processing on data objects immediately.

Examples for window-less blocks

Reader: read data from disk or network, create objects and send them on. Two versions of the file reader exist: A parallel one that internally uses up to nine threads to create message objects and a single-threaded one.

Statistic: calculates mean values or distribution statistics for time intervals.

Dumper: writes object attributes to disk, e.g. as CSV file.

Examples for window-based blocks

Sorter: sorts data according to start or end time. The window moves according to the timestamps of received data. Stored data with timestamps smaller than the lower window edge is forwarded. Data received with timestamps smaller than the lower window edge is dropped.

Joiner: combines records of the same flow that have been exported separately due to timeouts. A window specifies the *maxDuration*, i.e. how long the block should wait for another record before expiring and forwarding the *JoinedFlow*. *maxWaitingTime* specifies the maximum length of the created JoinedFlows. Without the second parameter JoinedFlows of flows lasting for a very long time would be forwarded to downstream blocks very late.

Correlator: with more than one input these blocks correlate different data streams, e.g. based on timestamps. Typically, timestamps of the data compared are not exactly equal or have an offset resulting from measurement. Thus windows are necessary.

While processing times of records in window-less blocks are rather fix, this time is highly variable for window-based blocks. In a window-based block, a data object can either be dropped or kept (added to internal data structures). Additionally, it can lead to window movement and thus to the expiration of several objects. This leads to a high jitter in processing time and makes buffering between window-based blocks and other threads necessary. Without any or with only small queues, window-based blocks are likely to stall the chain.

3.3 Thread and message configuration parameters

Our framework allows us to configure whether a processing block runs as an independent thread or not. A block that does not run as a single thread belongs to the thread of the upstream block and gets the control flow when it receives data. Obviously, this results in the constraint that readers must always run as a thread since they are the data sources. In correlator blocks it depends on the data from which input the next object is read. Thus they always run as a thread, since using the control flow from upstream block would drastically increase complexity. Using more threads helps exploiting modern multicore architectures, but also comes at the cost of higher memory consumption due to objects present in queues that are used to connect blocks running on different threads. The concept of thread pools is not applicable, since its purpose is to reuse a limited number of existing threads instead of creating them for each arriving task.

We realized that the high number of objects flowing through the blocks leads to a high context switch rate and high CPU time in the operating system (OS). Since Java maps threads directly to kernel threads, the OS is involved in locking and context switch operations. Thus, each object added or removed to queues possibly involves a switch from user mode to kernel mode and back. To mitigate these effects, we introduced burst messages, where several data objects are sent in one message. Due to performance reasons, burst messages are only applied for message exchange between threads. The number of included objects is called burst size. Queue operations happen less often and it is more likely that threads can run for a longer time without being blocked. However, this also comes at the cost of additional memory consumption. The size of burst messages is configurable. We study its impact in the next section.

4 Performance Evaluation

4.1 Measurement Scenario

For performance evaluation we selected a reference processing chain (Fig. 3). The chain reads Flow Records from two routers from files, aggregates Flow Records of

the same flow in a joiner and correlates these *JoinedFlows*. Statistics evaluate the time, byte and packet difference of *JoinedFlows*. This allows to extract packet loss rates, byte count inaccuracies and network delay ([3]). Where processing blocks require sorted data, sorters are employed. Due to the high number of window-based processing blocks, this chain requires a considerable amount of memory and is thus suitable to study memory effects.

Our performance evaluation centers on the throughput. We try to identify influencing factors by studying system time, user time and the time the garbage collector needs.

For measurement the depicted chain processed two uncompressed files with 65 million Flow Records in total. Each file contained records from only one exporter. The queues between threads had a size of 100 messages. According to the characteristics of our data we set the windows of the blocks as follows: Sorter 1,2: 20 s; Joiner 1,2 *maxWaitingTime*: 5 min; JoinedFlowSorter 1,2: 5 s; last Sorter: 3 s. The measurements were performed on an Intel Xeon X3360 quad core 2.83 GHz processor with a total amount of 8 GB memory. The software configuration included Sun JVM version 1.6 update 15 on Ubuntu 9.10 64 bit with kernel version 2.6.31. The standard garbage collector was used. Measurements with different garbage collector settings showed similar results. The only configuration done with the JVM, was to set the initial and maximum heap size to the same value. Each measurement was done five times. In charts we show the mean values of the results with error bars showing the minimum and maximum absolute value. We measured the real time r, the system time s and the user time u with the `/usr/bin/time` program. The time the garbage collector needs was obtained with the `GarbageCollectorMXBean` class provided by the JVM. This time is included in u.

Several parameters likely influence the throughput behavior. The throughput can be increased by using faster CPUs with a larger main memory. We also did measurements on a machine with two quad core Opteron CPUs and 48 GB main memory, where we observed a speedup of up to factor two. We also examined other processing chains and we found proved that other processing chains show a similar behavior. In this work we will focus on the thread assignment and the size of burst messages.

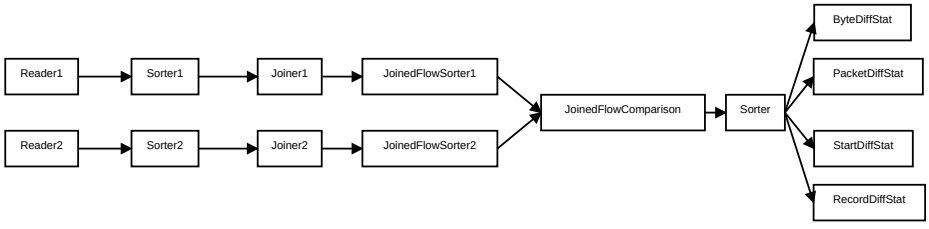

Figure 3. Joined Flow comparison chain

thread assignment	independent threads
A_1	Reader1 (single), Reader2 (single), JoinedFlowComparison
A_2	Reader1 (parallel), Reader2 (parallel), JoinedFlowComparison
B	Reader1 (parallel), Reader2 (parallel), JoinedFlowComparison, Sorter
C	Reader1 (parallel), Joiner1, Reader2 (parallel), Joiner2, JoinedFlowComparison, Sorter
D	Reader1 (parallel), Sorter1, JFSorter1, Reader2 (parallel), Sorter2, JFSorter2, JoinedFlowComparison
E	Reader1 (parallel), Sorter1, JFSorter1, Reader2 (parallel), Sorter2, JFSorter2, JoinedFlowComparison, Sorter
F	each block is a thread (parallel readers), 14 threads

Table 1. Thread Assignment Patterns

4.2 Benefit of multithreading

In the following we show which assignment of threads to processing blocks makes sense and how the framework benefits from multithreading. The decision to combine processing blocks needs to be based on their characteristics like the needed processing time, IO intensity or the number of exchanged messages. In general, it is a good solution to combine several processing blocks if their tasks are simple.

We study the seven assignment patterns listed in Tab. 1. All processing blocks that are not listed run in the same thread as their predecessor. We used configurations with the minimum number of threads (assignment A_1) up to the maximum number (assignment F). The patterns B to E try to split the long chains into shorter sub-chains. The parallel reader is the reader with several internal threads, the single reader is the single-threaded version.

Fig. 4 shows the throughput in Flow Records per second and the CPU utilization $\rho = \frac{u+s}{r}$ related to different thread assignment patterns. As we can see, throughput as well as utilization are nearly independent of the thread assignment.

The difference between A_1 and A_2 appears mainly in the utilization. But optimizing the utilization is not our goal, since we want to achieve a high throughput. On the basis of the results we could see, that u as well as r is in both cases almost the same. Only s is in A_1 greater, because of blocking since the single-threaded reader cannot create message objects fast enough so that the following threads can work continuously.

In pattern C a lot of data must be processed in the first thread containing the reader and the first sorter. From the result we can see, that blocks processing a high data rate should run as an independent thread.

The performance decreases in F, because the memory consumption increases, especially with higher burst sizes. The reason is that more data must be kept in

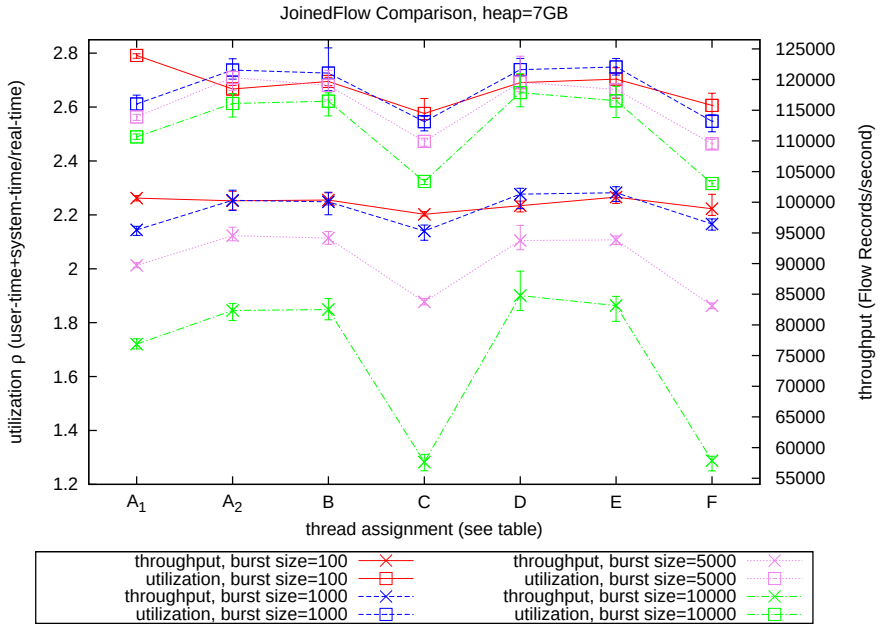

Figure 4. Benefit of multithreading with different thread assignments

the queues between the threads. Also the garbage collector must run more often to free memory.

One reason for the small overall utilization is the garbage collector, which stops the execution of the program to perform major collections. Because the standard collector is used, these collections are only done by one CPU core. The collection can be observed when running with low heap size: after several seconds of high utilization (about 3.8), during collections the utilization drops down to 1 for a few seconds. From these measurements we conclude that the burst size has a high influence on the throughput. Therefore, we study the impact of the burst size more detailed in the next section.

4.3 Impact of burst messages

Adding and removing messages to and from queues is expensive since locking operations and calls to the OS are required. This leads to high context switch rates and high s if the message rate is high. Atomic operations for locking and switches to the OS and back as well as switches between threads can be reduced by employing burst messages.

The more messages are aggregated into one burst message, the lower s becomes and so the context switch rate. On the other hand, using burst messages with bigger sizes results in a higher memory consumption. Thus a trade-off between the context switch rate and the memory consumption must be found.

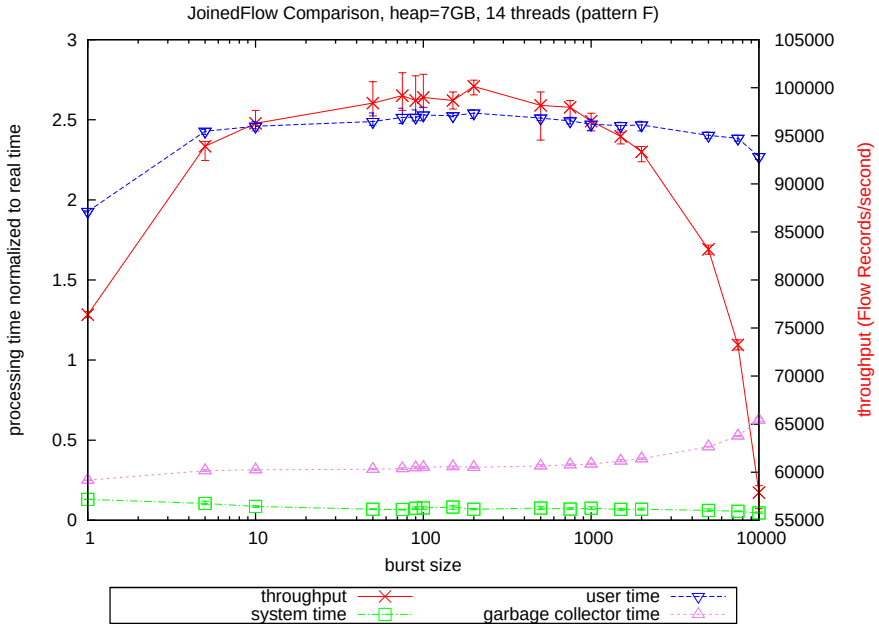

Figure 5. Impact of burst messages

Fig. 5 shows the throughput related to the burst size for thread assignment F. As expected, the throughput increases with increasing burst size. Interesting is the fact, that the normalized user time is more or less constant regardless of the configured burst size while the throughput increases. The reason is that the time axis is normalized to the real time. Thus it shows a kind of utilization. As we see from the results, the burst size should be greater than 100, otherwise the throughput is decreased by up to 25%. Much larger bursts do not lead to better performance, but eventually to performance degradation by up to 50% due to the high memory consumption of larger burst messages. As another effect the increasing garbage collection time can be seen. The reason is the above mentioned higher memory consumption of greater burst messages, so that the garbage collector has to free more memory.

With increasing burst size there is a decrease of s from 1.3% to under 0.5% of the real time. Considering the amount of lock operations required on the FIFO queues, we expected a direct relation of s to the burst size. This cannot be observed. One reason might be the Linux mechanism of fast user space mutual exclusion (futex) [11], which we have found out by using `strace`. So not every access to a FIFO queue results in a context switch, because the mechanism tries to do synchronization in most cases in user space without switching into kernel mode. While this mechanism is cheaper than kernel space semaphores, it is still costly due to atomic operations. Thus, the high locking frequency with low

burst sizes has still impact on throughput and contributes to the performance degradation shown in Fig. 5.

5 Conclusion

We presented a framework for processing of Flow Accounting data in Java. The performance evaluation showed that on multicore architectures, using more threads than cores is beneficial despite the additional memory required. Additionally, the usage of burst messages further speeds up processing since less operating system interactions are required. We investigated the impact of the burst size and showed that from a certain size the memory consumption and therefore the overhead introduced by garbage collection reduces throughput. The framework seems to be suitable for online processing of Flow Accounting data received directly from the network.

References

1. Claise, B.: Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational) (October 2004)
2. Claise, B.: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard) (January 2008)
3. Kögel, J.: Extracting performance metrics from netflow in enterprise networks. Second Workshop on the Usage of NetFlow/IPFIX in Network Management (October 2009)
4. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, New York (2003)
5. Chandrasekaran, S., et al.: Telegraphcq: Continuous dataflow processing for an uncertain world. In: CIDR. (2003)
6. The CoMo project. http://como.sourceforge.net/
7. Dübendorfer, T., Wagner, A., Plattner, B.: A framework for real-time worm attack detection and backbone monitoring. In: IWCIP '05: Proceedings of the First IEEE International Workshop on Critical Infrastructure Protection, Washington (2005)
8. Marinov, V., Schönwälder, J.: Design of a stream-based ip flow record query language. In: DSOM '09: Proceedings of the 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Venice (2009)
9. Ciccarese, P., Larizza, C.: A framework for temporal data processing and abstractions. (2006)
10. Spring Framework. http://www.springsource.org/
11. Franke, H., Russell, R., Kirkwood, M.: Fuss, futexes and furwocks: Fast userlevel locking in Linux. In: The Ottawa Linux Symposium. (2002)