# Processing of Flow Accounting Data in Java: Tool 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 concept 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 tool for processing of Flow Accounting data written in Java. It provides processing blocks for aggregation, sorting, statistic, correlation and other tasks. Besides reading data from files for offline analysis, it can also process data directly received from the network. In terms of multithreading and data handling, the tool is highly configurable in order to optimize its performance for a given task. For setting these parameters there are several trade-offs concerning memory consumption and processing overhead. In this paper, we 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 collected in routers and exported 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 serve as input for anomaly detection or extraction of network characteristics.

Processing of Flow Accounting data is challenging, since in large networks routers export several hundred million flow records per hour. Basically, three common approaches can be distinguished to handle and process this data. First, flow records can be stored in a central or distributed database for creating reports and doing 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 to files for offline analysis without a database. Third, flow records can be analyzed online directly in memory without storing the flow records themselves.

We developed a flow processing toolbox 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 toolbox processes flow records in a streaming fashion, i.e. data flows through a chain of processing blocks and each block can keep data using a sliding window for performing processing tasks. For tasks like joining and sorting flow records, the window size depends on characteristics of the Flow Accounting data. These result from router configuration and traffic characteristics. Larger windows increase memory consumption. However, this is no sensible metric in our case due to garbage collection. Rather more memory consumption results in more garbage collector overhead and thus reduced throughput. This paper studies these dependencies for giving hints on optimal settings for the toolbox. Besides this, the processing blocks can be assigned to different threading schemes, enabling the exploitation of 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 toolbox, 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 based on the key information on each flow is stored. The router updates the flow information (e.g. byte and packet count) either for each packet seen (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 several Flow Records in a packet to the collector exist. Most often Cisco system's NetFlow format is used. The current NetFlow version 9 [1] and its successor IPFIX [2] is a flexible format based on templates, while the fixed format of version 5 dominates current deployments. Since the data is sent via UDP, packets containing several records might be lost.

For determining the export time $t_x$ of a flow record, there are several strategies in Cisco routers. *Inactive timeout*: if for a flow there is no more packet seen for $\Delta t_{inact}$, the flow is exported. *Active timeout*: if a flow was active for a time period greater than $\Delta t_{act}$, the flow is exported. *Fast timeout*: If after $\Delta t_f$ a

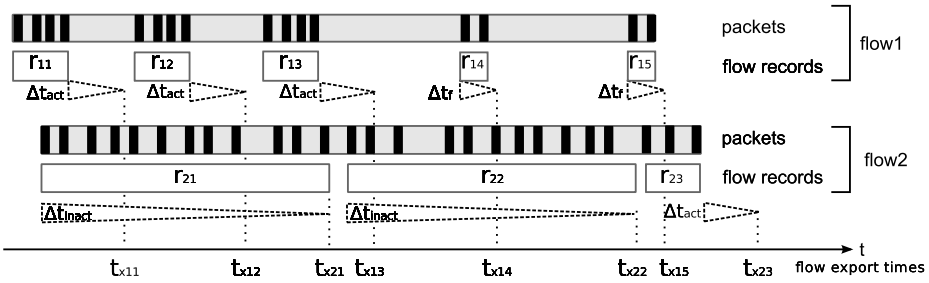


**Figure 1.** Flow export: timers of different strategies and resulting record order

flow contains less than $n_{fast}$ packets, the flow is exported. *Cache clearing*: if the cache becomes full, the router exports flows earlier than defined by the timeouts. Determining flow ends by tracking TCP connection state is implemented only in a fraction of small routers. The timeouts are illlustrated in Fig. 1 for two flows. We can see, that information on flows can be distributed across different records with long breaks in between band 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 several hundred Million Flow Records arrive at the collector per hour. Thus, processing or storing this amount of data is challenging. A common approach for offline analysis is using load balancers that distributed the traffic across several collectors that form a distributed database. There are several commercial tools that collect and analyze Flow Accounting data in that way and in most cases data is aggregated (e.g. on time intervals) for reducing storage requirements. Thus, evaluations that need fine grained timing information are not possible. For collection and basic processing of Flow Accounting data, several free tools such as *flowtools* and *nfdump* as well as DB-based reporting and analysis tools (e.g. *nfsen*) exist.

Evaluation algorithms that rely on fine grained information are typically processing and memory intensive, thus a DB based tool is not feasible. Additionally, we want to correlate different network characteristics or data from different data sources, which is not possible using current tools. There are several approaches for processing network monitoring data in a streaming fashion, which is close to the domain of data stream management systems (DSMS). Related tools are Gigascope [3] for packet trace processing, the TelgraphCQ [4] DSMS or the network monitoring specific CoMo project [5]. Other tools focus on the dispatching of NetFlow UDP packets [6] or on Flow Query Languages [7]. A data processing pipeline in Java is presented in [8]. Its focus is on processing large objects (e.g. images), thus it is unfeasible flow accounting data.

## 3 Data Processing Framework

### 3.1 Architecture

We developed our processing framework for processing of Flow Accounting data, especially for fine grained analysis of flow records and extraction and correlation of network metrics. This includes tasks like aggregation of values and records, matching records from different sources and calculation of derived metrics. In order to handle the huge amount of data, we use a stream-based approach that capitalizes on modern multicore architectures.

Our framework builds on interconnected processing blocks that form a data processing chain. Processing blocks exchange messages containing references to objects and are derived from a couple of generic blocks (Fig. 2). Each block has at least to implement a data source or a data sink interface, while all combinations with several interfaces are also possible. In Fig. 2 we can see an exemplary chain, where two `SourceBlocks` read NetFlow data from files or the network and forward the data to `SinkSourceBlocks` that do data aggregation to `JoinedFlows`. The two pipelines are merged in a `CorrelationBlock`, which creates correlation result objects that are statistically evaluated in three `SinkBlocks`.

Processing chains are constrained to a directed acyclic graph. While cycles could make sense for e.g. 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 as different threads, they are connected via blocking FIFO queues, as shown in Fig. 2 for the configuration with the maximum amount 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 by concurrent access in parallel paths by not modifying objects after they left the block where they have been created. Due to garbage collection provided by the Java Virtual Machine (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.

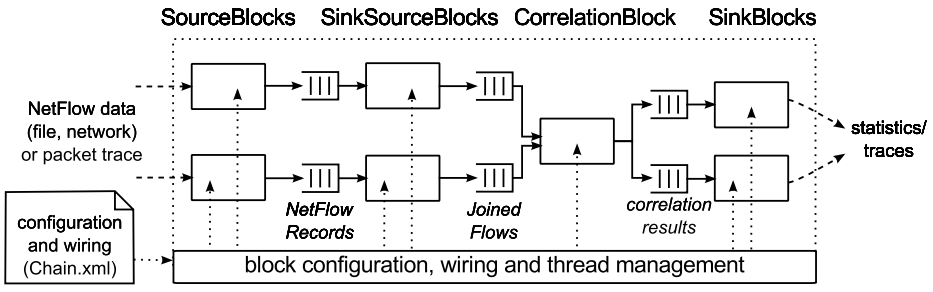


**Figure 2.** Exemplary processing chain showing basic types of processing blocks

At startup, the chain is set up by a central component that also does thread management. Configuration is based on an XML file that describes processing block parameters and chain structure. For this, we build on the dependency injection mechanisms provided by the Spring Framework [9]. After all objects are created and wired as defined, threads are started and the sourceblocks will start 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 performend by deregistering connections from upstream blocks, which will then shutdown.

### 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 the network, create objects and send them on.
**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 time-based 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 the `JoinedFlow`. `maxWaitingTime` specifies the maximum length of the created JoinedFlows. Without the second parameter JoinedFlows of flows lasting over 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. on timestamps. Typically, timestamps of the data compared are not exactly equal or do have an offset resulting from measurement, thus windows are necessary.

While processing times of records flowing into window-less blocks are rather fix, this time is highly variable for window-based block. In a window-based block, a data object can be either dropped, kept (added to internal data structures), or lead to the expiration of several objects due to window movement. This effect leads to a high jitter in processing time and makes queues between window-based blocks running as single threads and other threads necessary. Without or only small queues, there is a higher probability that window-based blocks stall the pipeline.

### 3.3 Thread and message configuration parameters

In our framework we can configure whether a processing block runs as an independent thread or not. If it does not run as a single thread, it 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 always must run as a thread since they are the data source. Also correlator blocks run always as a thread, since it depends on the data from which input it reads. Using the control flow from upstream block would drastically increase complexity in this case. Exploiting 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 does not apply here, 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 consumption 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. The number of included messages is called burstsize. Thus, 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 and its impact is studied in the next section.

## 4 Performance Evaluation

### 4.1 Measurement Scenario

For performance evaluation we selected a reference processing chain (Fig. 3). The chain takes flow records from two routers, 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 derive packet loss rates, byte count inaccuracies and network delay. 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.

Our performance evaluation mainly covers the throughput of the program. In addition to that, we try to find what impacts throutput by studying system time, user time and the time the garbage collector needs. We considered the impact of memory available by assigning different amounts of heap memory to the JVM.

For the measurements each reader in the depicted chain processed a flow-tools file with 6 million flow records in total. The files were pre-filtered and uncompressed, so that they contained only records from one exporter. The queues between threads had a size of 100. According to the characteristics of our data

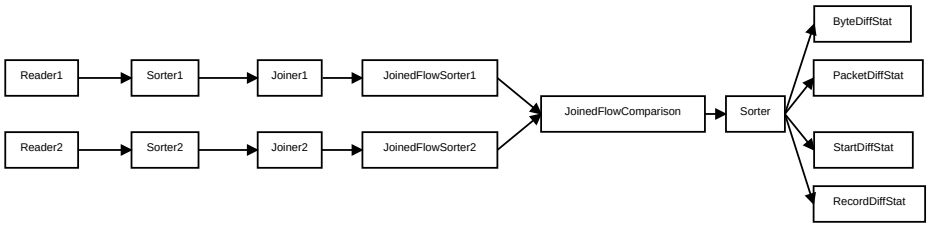


**Figure 3.** Joined Flow comparison chain

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 following measurements results were obtained on a Intel Xeon X3360 quad core 2,83 GHz processor with a total amount of 8 GB memory. The Sun JVM version 1.6 update 15 was used with Ubuntu 9.10 64 bit, Kernel version 2.6.31. The standard garbage collector was used. The only configuration done with the JVM, was to set the initial and maximum heap size to the same value. Each measurement was done twice. We did measurements of 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 measured with the `GarbageCollectorMXBean` class provided by the JVM. This time is included in $u$.

To study the throughput behavior we can vary different parameters. We will focus on the thread assignment, the size of burst messages and the available heap memory. Of course the throughput can be increased in using faster CPUs with a bigger main memory. We did also measurements on a machine with two quad core Opteron CPUs and 48 GB main memory, where we observed a speedup of up to factor 2. Besides other hardware we also examined other processing chains, so we can proof that other processing chain showed a similar behavior.

| thread assignment | independent threads |
|---|---|
| A | Reader1, Reader2, JoinedFlowComparison |
| B | Reader1, Reader2, JoinedFlowComparison, Sorter |
| C | Reader1, Joiner1, Reader2, Joiner2, JoinedFlowComparison, Sorter |
| D | Reader1, Sorter1, JFSorter1, Reader2, Sorter2, JFSorter2, JoinedFlowComparison |
| E | Reader1, Sorter1, JFSorter1, Reader2, Sorter2, JFSorter2, JoinedFlowComparison, Sorter |
| F | each block is a thread, 14 threads |

**Table 1.** Thread Assignment Patterns

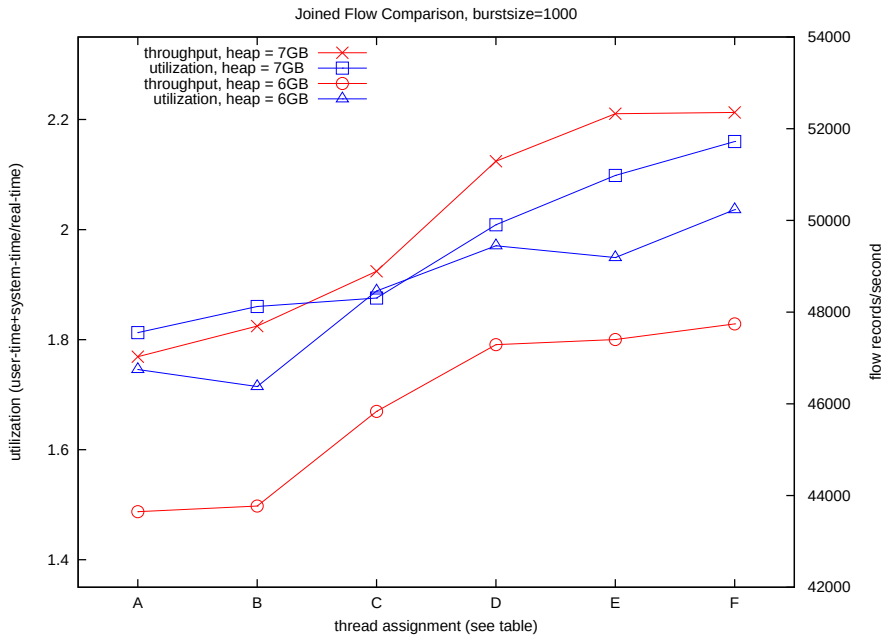


**Figure 4.** Benefit of multithreading with different thread assignments

### 4.2 Benefit of multithreading

In the following we show which assignment of threads to processing blocks makes sense and how the toolbox benefits from multithreading. A decision which blocks should be combined can be based on several characteristics of a processing block, 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 six assignment patterns listed in Tab 1. All processing blocks that are not listed run in the same thread as their predecessor. We used chains with the minimum number of threads (assignment A) up to the maximum number (assignment F). The patterns B to E try to split the long chains into shorter sub-chains.

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 and two different heap sizes. One reason for the poor overall utilization is again 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. The collection can be observed during the program runtime. Between the normal processing, where the utilization is about 3.8, the utilization drops down to 1 for a while. Especially in the case with the smaller heap these pauses are compared to the normal execution relatively long.

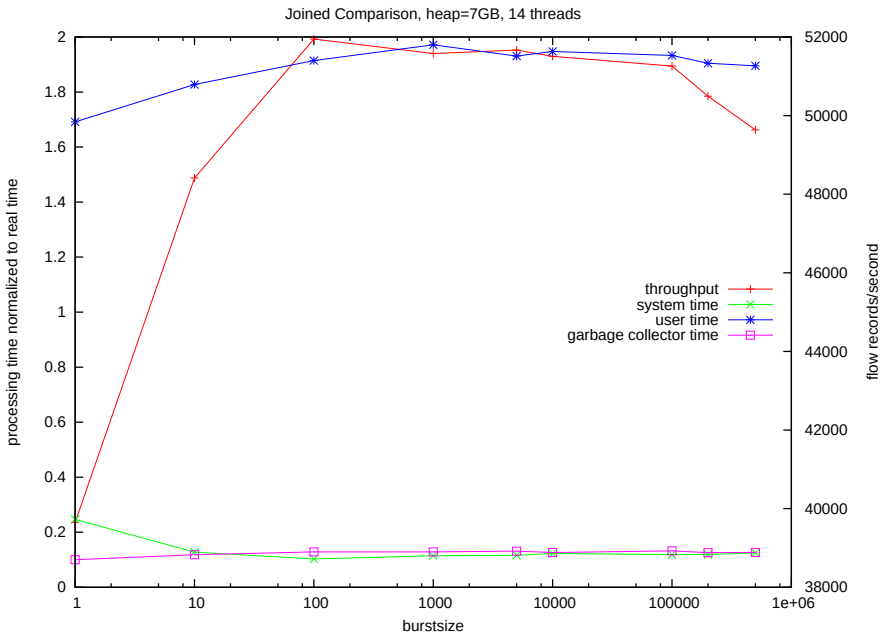


**Figure 5.** Impact of burst messages

The results show, that even using a higher number of threads than CPU cores is advantageous. However, it does not always make sense to use the maximum number of possible threads, especially with respect to memory.

### 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. Both, 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.

Fig. 5 shows the throughput related to the burstsize. As expected the throughput increases with increasing burstsize. Interesting is the fact, that the normalized user time is nearly constant regardless of the configured burstsize although the throughput increases. The reason is that the time axis is normalized to the real time. Thus it shows a kind of utilization. Furthermore we can see the desired decrease of system time. As we see from the results, the burstsize should

be greater than 100. Much larger bursts do not lead to better performance, but eventually to performance degradation.

However, we observed that $s$ is not directly related to the burstsize. One reason might be the uses of the Linux mechanism of fast userspace mutual exclusion (futex) [10], which we have found out by using `strace`. So not every access to an `ArrayBlockingQueue` results in a context switch, because the mechanism tries to do an resolution in userspace without switching into kernelmode to do locking on the shared memory.

## 5 Conclusion

We presented a tool 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 tool seems to be feasible 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. 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)
4. Chandrasekaran, S., et al.: Telegraphcq: Continuous dataflow processing for an uncertain world. In: CIDR. (2003)
5. : The CoMo project. http://como.sourceforge.net/
6. 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)
7. 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)
8. Ciccarese, P., Larizza, C.: A framework for temporal data processing and abstractions. (2006)
9. : Spring Framework. http://www.springsource.org/
10. Franke, H., Russell, R., Kirkwood, M.: Fuss, futexes and furwocks: Fast userlevel locking in Linux. In: The Ottawa Linux Symposium. (2002)