The Journal Pattern for Streaming Data

Antonio Maratea[1], Angelo Ciaramella and Rita Perna

Department of Science and Technologies, University of Naples "Parthenope"
[1] *corresponding author*

Abstract

In the distributed streaming data processing scenario, most of the frameworks implement minimal variations of the Publish-Subscribe pattern, where message passing happens directly between each Publishers and the group of its Subscribers. This work introduces a novel pattern, named *Journal*, that exploits a so called *Editor* for filtering or modifying the data stream in a principled manner. The Editor can be integrated into the Publish-Subscribe pattern with two different schemata, and has been used to implement multiple subsampling strategies, so to reduce the volume of the forwarded data, create new communication channels and match the ingestion capacity of the consumers. An actual test using Apache Kafka with a stream of simulated data has confirmed the viability of the Editor integration into Pub-Sub. We evidence that with the *Journal* pattern the risk of saturation of a channel can be significantly lowered and the latency of processing from clients can be notably reduced. We stress that the *Journal* pattern is very general and can be extended to multiple other purposes.

Keywords

Streaming data, Design Patterns, Subsampling, Publish/Subscribe, Apache Kafka

1. Introduction

Streams coming from sensors, IoT, from online transactions or monitoring systems, satellites or high frequency trading are ubiquitous and relative data grow exponentially, actually representing a significant portion of the Big Data phenomenon [1, 2, 3]. At the same time the resurgence of Event Driven Applications in form of reactive programming, pushed by strict scalability requirements and large scale infrastructures availability, has made event streams and streaming frameworks more popular than ever before [4, 5, 6, 7].

While many frameworks exist to manage multiple sources of streaming data and multiple clients, most of them are based on the *Publish-Subscribe* pattern, where a so-called *broker* forwards the messages/data/events published on a channel/topic to all the clients that subscribed to that channel/topic. The most common behaviour of such systems is to forward all published messages/data/events on a channel to each subscriber/client, regardless of the ingestion capacity of the client itself, often resulting in an unacceptable lag in the analysis, or in channel overload, or in a subscriber crash, or in any combination of these, finally slowing down the entire system [8].

ITADATA2022: The 1st Italian Conference on Big Data and Data Science, September 20–21, 2022, Milan, Italy
antonio.maratea@uniparthenope.it (A. Maratea); angelo.ciaramella@uniparthenope.it (A. Ciaramella); rita.perna001@studenti.uniparthenope.it (R. Perna)
0000-0001-7997-0613 (A. Maratea); 0000-0001-5592-7995 (A. Ciaramella)
© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3340/paper63.pdf

In the following first some classic patterns of communication and message-passing will be described in general and compared, targeting data stream modeling, then a new pattern, named *Journal*, will be proposed in two variations, both able to account for the ingestion capacity of the clients [9]. While a specific strategy will be proposed through this pattern to solve the ingestion problem, it turns out as a very general and reusable tool, open to many possible extensions.

The paper is organized as follows: in Section 2 background information is discussed and in Section 3 the novel *Journal* pattern is introduced, its variations discussed and an example application presented. Conclusions are drawn in Section 4.

2. Background

The idea of a *pattern* as a recurring and recognizable structure of the real world well anticipates its discovery and application to software design and software architecture. While it has been found profitable and fruitful in many concrete domains, like urban design [10], it must be stressed that the mother of all patterns is just one — abstraction — and in this sense all discovered patterns are nothing more than predefined solutions abstracted from specific instances of common problems. While it is a matter of debate how general these solutions are or how limited is the number of actually new problems that computer scientists continuously face up, it is not a surprise that software design patterns emerged during the Object-Oriented Programming (OOP) wave of the nineties, with its accent on reusability and modularity [11]. Depending on the scenario (the chosen level of abstraction, the chosen paradigm and language, the data representation, the specific set of constraints or even the hardware architecture with which a common problem is tackled), some design patterns may become inapplicable or simply disappear because they end up to be transparently embedded in the chosen tools. It must be stressed that a problem changes changing its representation and the tools used to tackle it, so a pattern may not be suitable to be applied on any instance of the problem that in theory it should solve; at the same time this is not a proof of its inconsistency, rather of the great variability of problem instances, of the somewhat inherently limited abstraction of the pattern itself, and of the probably underestimated number of variations for the same problem that modeling choices generate.

A *design pattern* can be defined as a general modeling solution that abstracts from the specific instance of a problem and can be used to model other instances of the same problem in different contexts. A *software design pattern* can be defined as a general modeling solution to a problem in a given software scenario (paradigm, language, level, data representation, hardware, etc.), whose purpose is to use abstraction to recognize and model different instances of the same problem in different scenarios and promote code reusability and "best practices"[1] spread.

The formalization of this concept in the realm of OOP is due to the infamous Gang of Four (GoF) book [12], which defined 23 patterns into three different categories: creational, behavioral and structural. The book is still generating research and interest (please see [13] and references therein for a survey).

[1] By the end patterns are design tools. Like any tools they can be used for the good and for the bad, the myth that using a pattern is by itself a good practice should be dispelled.

2.1. Messaging Patterns

The one-way communication among different classes in the simplest case is Point to Point (P2P) asynchronous and is manages by a shared queue of non-zero capacity, where the sender is called *producer* and the receiver is called *consumer*. This schema generates the well known *bounded buffer* (or producer/consumer) synchronization problem, where blocking primitives act on both sides (on the producer and on the consumer), slowing down the producer when the ingestion capacity of the consumer does not match or exceeds the volume of produced data or when the consumer is not always active. Multiple producers and multiple consumers retain and somewhat worsen this side effect. On the opposite side, choosing message passing schemata that do not use a shared queue allows an efficient many-to-many communication among more loosely coupled consumers, were producers are never blocked, increasing scalability and efficiency. When considering the so-called Event-Driven Architecture (EDA from now on), messages to be communicated are events (instead or raw data) and the message passing schema is crucial for the proper functioning of the application, that basically only react asynchronously to the propagated events. This schema is non blocking, highly decoupled and scalable, better supporting demanding distributed and cloud-native applications.

In the following, the Observer, Broker, Mediator, Proxy and Pub/Sub patterns are compared, due to their similarity, possible ambiguity, and relevance for messaging and data stream processing.

- The *Observer* defines a one-to-many dependency between an observable class and different observers classes. The observable keeps the list of of all its observers and sends a push notification to all of them whenever its state changes. It includes a subscription mechanism into the observable object, such that each observer can subscribe or unsubscribe to the stream of events coming from that object.
- First of all, the term "broker" has a generic meaning that should not be confused with the pattern here described, that refers specifically to the EDA topology and the propagation of messages/events. A *Broker* can be seen as a chain of Observers, where the first Observed object generates the event, propagates the event to its subscribers, and each of them starts its own processing and eventually propagates the event to its relative subscribers. There is no control of the order of execution of the various operations triggered by the event.
- The *Mediator* is grounded on the idea of decoupling different objects that have to communicate with each other, redirecting messages and hence both coordinating actions and hiding the senders from the receivers. Compared to the Broker, when used in EDA, it allows to control the sequence of consequences of an event. Compared to the Observer, the observable object — i.e. the one that generates the message/event — is not aware of its observers, nor should it manage their subscriptions.
- *Proxy* again is a term with a generic meaning that should not be confused with the pattern here described, that refers specifically to the GoF book [12]: it is a structural design pattern that provides a placeholder for another object, in order to provide transparently meta functionalities (like caching, logging, access control, checking preconditions, etc.) to the target object. The target object an its proxy share the same interface, so it is transparent for the client. It is not to be confused with the generic idea of a proxy that should be modeled through a Mediator.

- The *Publish-Subscribe* (or simply Pub-Sub) is a messaging pattern suitable for many-to-many communication in large-scale systems [14], in which multiple *Publishers* communicate with multiple *Subscribers* though a set of predefined *channels* (or *topics* in Apache Kafka terminology), much like a subscription to a physical journal in the real world. Messages are grouped in channels/topics and made persistent on a separate broker. Please note, this has nothing to share with the broker pattern described above.

Concerning the last pattern, Publish/Subscribe, it allows three different types of subscription:

- Pull-based: the Subscriber decides when to receive the data, performing a continuous polling cycle on the source. In this specific case, propagation is said to be demand-driven, that is driven by the demand for a new value.
- Push-based: in contrast, when the source publishes new data, it pushes them to each Subscriber. In this case, propagation is also called data-driven, meaning that when new data become available they will automatically be forwarded to the Subscribers.
- Push-Pull: it is an hybrid approach in which the Subscriber receives a push notification and, consequently, it will decide whether or not to accept the data (please see Fig. 2.1).

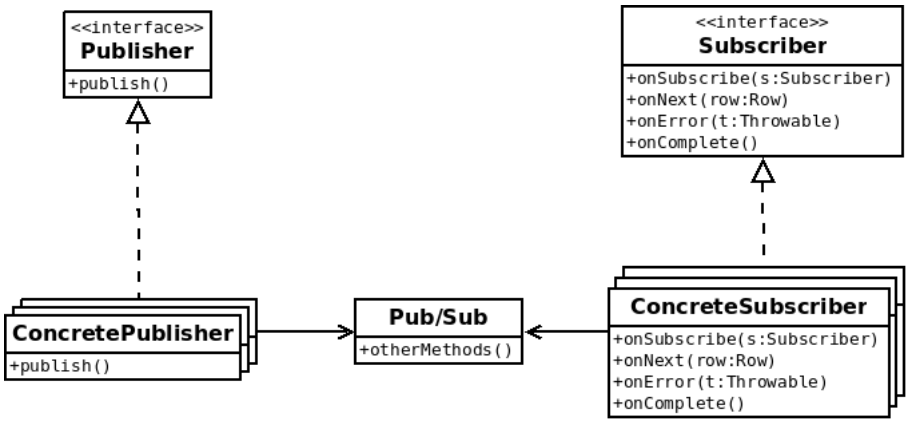

Figure 1: The Publish-Subscribe pattern where publisher push and subscribers pull the messages.

2.2. Data production and ingestion rates

The data production rate $\bar{p}$ of a software component $\mathscr{C}_i$ can be defined as the volume of data (in bytes) that in average this component produces in the unit of time. For a given time interval Δt, the average volume of produced data is given by the following formula:

$$\bar{P}_i = \bar{p}\Delta t \tag{1}$$

On the other side the ingestion capacity $\bar{c}$ of a software component $\mathscr{C}_j$ can be defined as the volume of data (in bytes) that in average this component is capable to process in the unit of

time. For a given time interval Δt the average volume of ingested data is given by the following formula:

$$\bar{C_j} = \bar{c}\Delta t \tag{2}$$

Given a software system $\mathcal{S}$ with n producers and m consumers on the same channel two observations must be made: first the volume of incoming data may be extremely variable over time and in general is a function of time t, that is $\bar{P_i} = f(t)$; second the ingestion capacity is affected by the chosen communication schema, and does not change continuously with time unless the application is deployed over an elastic cloud.

If reads by the consumer are blocking, to have a responsive system a necessary condition is:

$$\sum_{i}^{n} \bar{P_i} \ll \sum_{j}^{m} \bar{C_j} \tag{3}$$

The strong constraint being due to the need of sizing a system on the peaks of load. This is not a sufficient condition in case the components are serialized (think to the bounded buffer above), so in the worst case

$$\sum_{i}^{n} \bar{P_i} \ll \min_{j} (\bar{C_j}) \tag{4}$$

that results in the ingestion capacity being the true bottleneck of the system. If (3) or (4) do not hold, Producers end up accumulating queues of data to be transferred that will never be processed.

Using asynchronous and parallel reads without blocking acknowledges speed notably thing up, to the point where the ingestion capacity affects only the lag of processing of the clients, not the rate of production or the volume of data in the channel. At this point the bottleneck becomes the processing latency of the consumers, that for any $\bar{C_j} < \sum_i^n \bar{P_i}$ goes to infinity. In case quasi-real-time answers from all the consumers are required, again equation (4) holds.

3. The *Journal* pattern

To have low latency and highly responsive large scale applications it is crucial to guarantee high throughput and to avoid unnecessary waits or bottlenecks in data processings [15]. Usually these are the result of some anti-patterns (aka "bad practices"), where unnecessary serialization, synchronous processing, or lock contention make the whole system wait for its slowest component to complete a given operations before continuing, and/or wrongly sized components have been deployed. As pointed out in the previous section, the ingestion capacity of data consumers is one of the crucial issues to be addressed. In this Section the *Journal* pattern is described. It is as an extension of Publish-Subscribe that involves a new component called *Editor* for filtering or modify directly the data stream. Two different integration strategies will be addressed hereafter.

3.1. The *Editor*

Much like a journal in the real world does not connect directly its readers to its writers, but filters and alters the published content, both to guarantee its quality and to adapt it to its

readership, a component called *Editor* dedicated to filtering or altering the produced data can be added to the communication schema of a software system.

In case of a data stream, the editor may be responsible for:

- reducing the volume of the data, using any sort of filtering, and/or compression, and/or deduplication;
- Enriching the data, hence increasing their volume, using upsampling, interpolation, data matching or metadata annotations;
- manipulating the data keeping their volume, for example sorting, shuffling or grouping them;
- removing unquestionable outliers;
- enforcing or casting the data into a common schema.

For example a filter may be used to remove a bias with a differential subsampling of the incoming stream on the base of the class of the instances before they reach a classifier or in alert systems only the peak value of an alert function in a given time window may be retained and forwarded, instead of all its values. More, depending on the approximation requirements and the regularity of the stream, only the average value within a given time window (fixed or variable in size) may suffice for the subsequent elaborations. Combining filtering and compression can easily reduce the volume of data by orders of magnitude relieving the data consumers from the risk of saturating their ingestion capacity, without significantly affecting the elaboration outcomes and dramatically decreasing their latency. On the other side, if the volume is not an issue the Editor can also enrich the data with the purpose of increasing precision or reusability.

In case of an event stream, the editor may be responsible for:

- removing duplicate events on the same channel that would produce multiple calls for the same function, or removing duplicate events among different channels if they end up with the same call. Both operations will reduce the volume of data and increase efficiency, even if the second scenario violates the principle of isolation of channels.
- removing obsolete or conflicting events, that is events that fire methods that override or mutually exclude each other, prioritizing the most recent;
- combining multiple simpler events that perform parts of a complex action into a single more complex event that calls a more complex function;
- adding meta functions like logging, caching or access control, much like the proxy pattern described above;
- monitoring for events anomaly, security breaches, component failures.

In this case subsampling is not a viable solution because messages represent events, not measurements, and cannot be skipped unless they are duplicated, obsolete, conflicting or anomalous.

In both previous cases, it must be stressed that the shorter the messages are, the better the system works. Any attempt to pass a BLOB as a message would produce an endless queue, huge waiting times and make the editor powerless.

3.2. Publish-Edit-Subscribe

There are at least two different ways in which the *Editor* can be integrated into the Pub-Sub pattern, none of which corresponds exactly to the patters described in section 2.1: these will be called the type 1 (open) and the type 2 (closed) variations.

Please note that there can be a many-to-many relation between channels and editors: multiple editors can act on the same channel, each implementing a different strategy to filter or alter the same stream, or an editor can subscribe to multiple channels, implementing a strategy to combine or match data from the different streams.

Type 1: the (semi) open journal

Here the *Editor* acts as a publisher from one side and as a subscriber from the other side: it reads all the messages on a channel (as a regular subscriber), filters or modifies these messages, and then publishes the filtered or modified messages on a dedicated channel built on purpose, as a regular publisher.

The original publishers continue to publish on the original channel (that can be made private), while the subscribers have access both to the original channels (if these are not private) or to the edited channels. All channels continue to live in parallel, and the subscription method can be parameterized in a way that automatically finds the appropriate channel for each specific subscriber (for example matching its ingestion capacity) or that it hides the channels that should be used only by the editor. This solution produces a proliferation of channels and a certain amount of overhead, but is able to remove the bottlenecks on the data ingestion discussed in the previous section, among other things. The editors are nor proxies, nor mediators, nor observers, nor brokers, as they have the responsibility of monitoring and modifying the stream and are not limited to redirecting it. Please see figure 3.2.

Type 2: the closed journal

Here the *Editor* acts as a *Broker* between the publishers and the channels, hence messages are filtered or modified before being published on their channel and subscribers are not aware of the editing process. While this solution has to maintain a different service, it does not produces channel proliferation, as the edited channels match the original ones. The editor receives the messages from the data sources, modifies them and forwards the modified messages on the target channel, as if it was itself the sole publisher. Please see figure 3.2.

3.3. Experiment

Hereafter an experiment is presented in which the *Journal* pattern of type 1 has been integrated into the Apache Kafka Pub-Sub framework and used to implement a data stream subsampler, able to match the ingestion capacity of the Subscribers and hence reducing their processing latency.

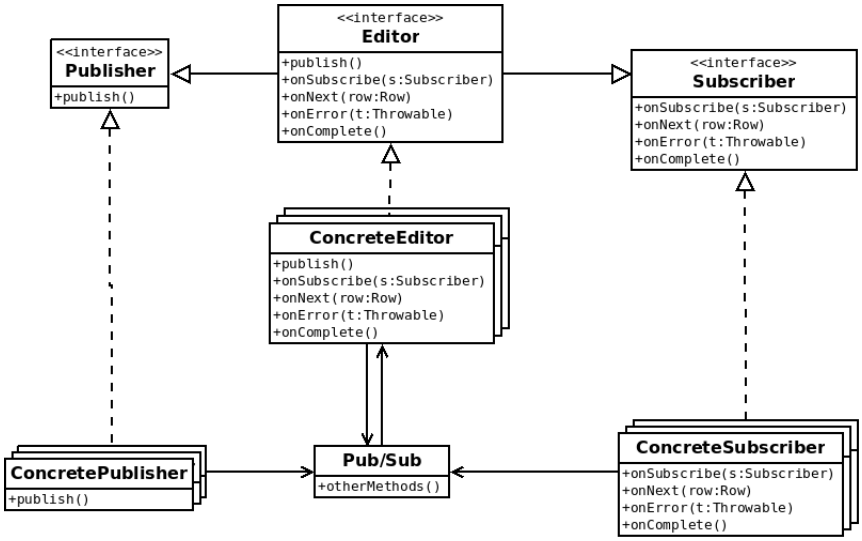

Figure 2: The *Journal* pattern type 1.

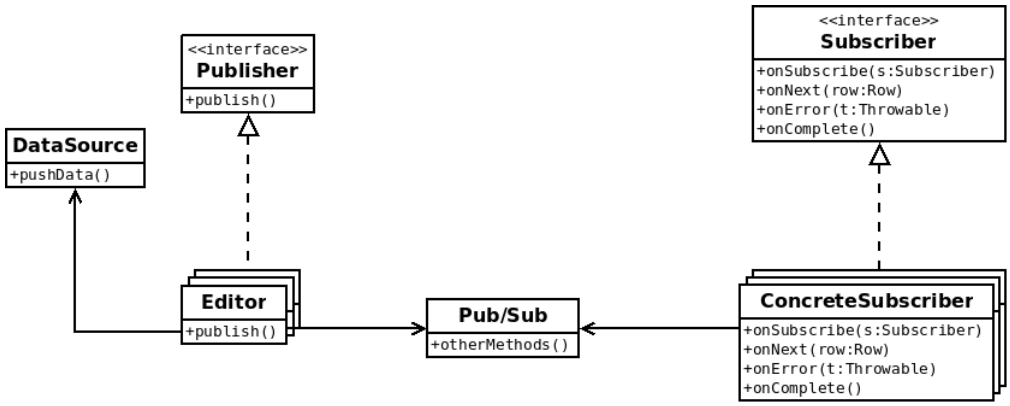

Figure 3: The *Journal* pattern type 2.

Apache Kafka

For sure there is a plethora of different implementations for the Publish/Subscribe pattern, but when looking for high scalability in a distributed environment, the Apache Kafka is with no doubts on the top-5 of the list. In the Kafka lingo the channels are called *topics* and the so-called "broker" [2] is the persistent queue of messages to be read, with a configurable retention policy. Kafka is pull-based, so it is not the broker that pushes the data to the Subscribers of a topic when they become available, rather the Subscribers continuously request the data from the broker through an infinite polling cycle querying the broker, who maintains a reading offset for each of them and returns all the records that they have not yet read. Once all subscribers have read a

[2]Nothing or very little to share with the *Broker* pattern described in section 2.1.

message and the configurable retention time has expired, messages are permanently deleted.

Kafka is designed for massive parallelism and optimized for a distributed environment, so any topic can be split into multiple partitions, each of which is an ordered persistent queue of some of the messages in the topic, with a configurable replication factor. Much like range partitioning in traditional RDBMS is used to split the records from the same table among different physical files according to a chosen criterium (hashing the primary key, for example), kafka partitions are physical queues allocated in different servers/brokers that act as persistent slots that receive the messages matching the chosen criterium (hashing a key, round robin, or with a custom partitioner). Partitions can be read in parallel by consumers, hence they allow the implementation of a load-balancing strategy and their proper use increases throughput (especially if serialization of reads or transactional confirmation of delivery are not mandatory).

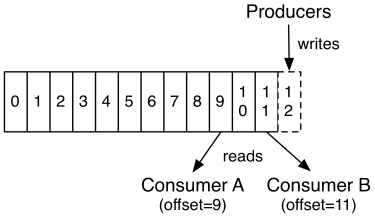

Figure 4: Offset representation.

As can be seen from figure 3.3, each of the Subscriber reading from the same partition of a given topic maintains its position based on the offset it has read. Thanks to the offset, the position of the next record to be processed by the Subscriber is determined.

The basic specification states that the record consumption occurs from the last committed offset (either the beginning or a specific offset) in a partition and proceeds with processing the records sequentially. However, this could be a significant limitation when the rate of messages produced by all Publishers in a topic/partition far exceeds the ingestion capacity of a Subscriber, as the processing of data by Subscribers can be delayed indefinitely and they end up being stuck, with a continuous accumulation of data to be processed. The resulting loss of information, failure to update data or delays in receiving answers in many applications are unacceptable.

A topic subsampler

As it has been clarified, one of the bottlenecks for a responsive application is the data ingestion capacity of the Subscribers, that in Kafka are pull-based and hence do not block the Publishers, but can saturate the persistent reading queue and delay indefinitely their processing outcome. As a mere demonstration of a possible application of the *Journal* pattern, a topic subsampler has been implemented.

When the stream of messages is composed of measurement data, for example data coming from a network of sensors, sampling theory gives an array of well grounded methods that allow to fulfill two apparently conflicting requirements: to reduce significantly the volume of the data and to keep the precision of the derived outcome [16]. Among the many subsampling strategies

that can be implemented, not necessarily static and not necessarily probabilistic, the simplest is systematic subsampling, that picks the elements with a reduced frequency $1/k$ with respect to the original signal, where k depends on the application (a probabilistic alternative in case a representative subsample of predefined size is required would be reservoir sampling). Thinking to the typical frequency of a sensor (it depends on the application, it can go from 1Hz to 10^8Hz), in most common cases for example for measuring the temperature of a room a frequency of 1Hz or lower would suffice, so subsampling the raw data stream can reduce the volume of data by orders of magnitude (another option is averaging, o selecting the peak value if the system is built to trigger an alarm).

In the performed experiment, Producers were built to read a toy dataset and generate multiple regular streams of random data, Topics were created in advance (some for the full stream of data and others for the subsampled streams) with a nominal peak throughput, finally multiple Editors where created with different nominal subsampling frequency, but the same strategy (systematic subsampling). The subsampling rate hard coded in the Editors was chosen to match the ingestions rate of the less eager Subscribers and named accordingly. The *subscribe()* method of the Subscribers/consumers were overloaded with a parameter representing their ingestion capacity and string pattern matching was used to choose the suitable Topic for each Subscriber. Empirical tests on the behavior of the partial average value have shown that the implementation works well, is responsive and represents an acceptable approximation of the original stream (please see figure 3.3). Depending on the specific use case, any strategy described in section 3.1 can be implemented through the Editor, without substantial modifications.

4. Conclusions

Data stream processing on large scale distributed systems needs to exhibit low latency and high throughput, regardless of the number of involved components. Most of the implementations of the Publish-Subscribe messaging schema are suitable for this purpose and can be configured to have very permissive message forwarding policies. Nonetheless, the ingestion capacity of the subscribers is an insurmountable hindrance to system responsiveness and should be addressed working directly on the data stream.

In this paper a new design pattern called *Journal* has been proposed as an extension of the Publish-Subscribe. It involves a new component called *Editor* whose purpose is to filter or modify directly the data stream, much like an actual editor of the real world does with an actual journal. Two different integration strategies have been proposed, one where the Editor is both a publisher and a subscriber and another where the editor is a broker.

To prove the viability and usefulness of the proposed pattern an actual test has been performed using Apache Kafka and implementing a filtering strategy through subsampling on a stream of simulated data. In this way the risk of saturation of a channel can be dramatically lowered and the latency of processing from clients can be notably reduced, matching their ingestion capacity. The proposed pattern is very general and can be extended to multiple other purposes.

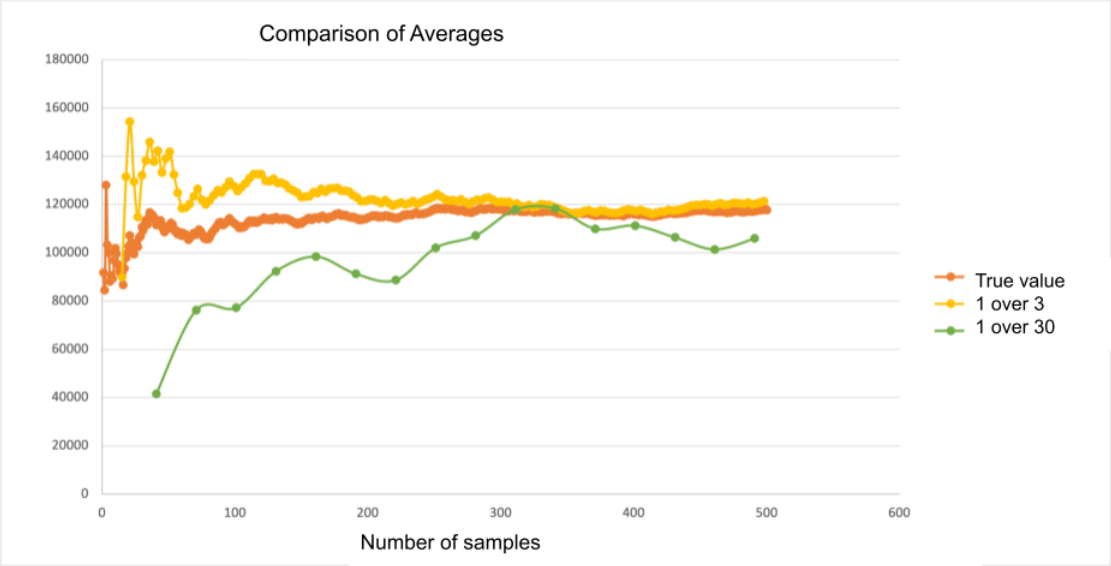

Figure 5: Comparison of the true average computed on all the samples of the stream (orange line) with the average computed on 1/3 of the samples (yellow line) and on 1/30 of the samples (green line). Toy data.

References

[1] A. Ciaramella, G. Giunta, Packet loss recovery in audio multimedia streaming by using compressive sensing, IET Communications 10 (2016) 387–392.

[2] A. Ciaramella, A. Staiano, G. Cervone, S. Alessandrini, A bayesian-based neural network model for solar photovoltaic power forecasting, Smart Innovation, Systems and Technologies 54 (2016) 169–177.

[3] E. Chianese, F. Camastra, A. Ciaramella, T. Landi, A. Staiano, R. A., Spatio-temporal learning in predicting ambient particulate matter concentration by multi-layer perceptron, Ecological Informatics (2019) 54–61.

[4] B. Engineer, A. Lombide Carreton, T. Van Cutsem, M. Stijn, D. Wolfgang, A survey on reactive programming, ACM Computing Surveys 45 (2012) 1–34. doi:10.1145/2501654.2501666.

[5] A. Tenorio-Trigoso, M. Castillo-Cara, G. Mondragón-Ruiz, C. Carrión, B. Caminero, An analysis of computational resources of event-driven streaming data flow for internet of things: A case study, The Computer Journal (2021).

[6] S. R. Goniwada, Event-driven architecture, in: Cloud Native Architecture and Design, Springer, 2022, pp. 241–294.

[7] S. Zhelev, A. Rozeva, Using microservices and event driven architecture for big data stream processing, in: AIP Conference Proceedings, volume 2172, AIP Publishing LLC, 2019, p. 090010.

[8] S. Khare, K. An, A. Gokhale, S. Tambe, A. Meena, Reactive stream processing for data-centric publish/subscribe, in: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, 2015, pp. 234–245.

[9] O.-C. Marcu, A. Costan, G. Antoniu, M. Pérez-Hernández, B. Nicolae, R. Tudoran, S. Bortoli, Kera: Scalable data ingestion for stream processing, in: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2018, pp. 1480–1485.

[10] M. S. Christopher Alexander, Sara Ishikawa, A Pattern Language: Towns, Buildings, Construction, "Oxford University Press", 1977.

[11] G. L. Fenves, Object-oriented programming for engineering software development, Engineering with computers 6 (1990) 1–15.

[12] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1 ed., Addison-Wesley Professional, 1994.

[13] A. Ampatzoglou, S. Charalampidou, I. Stamelos, Research state of the art on gof design patterns: A mapping study, Journal of Systems and Software 86 (2013) 1945–1964.

[14] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM computing surveys (CSUR) 35 (2003) 114–131.

[15] R. Simmonds, P. Watson, J. Halliday, P. Missier, A platform for analysing stream and historic data with efficient and scalable design patterns, in: 2014 IEEE World Congress on Services, 2014, pp. 174–181. doi:10.1109/SERVICES.2014.40.

[16] P. J. Haas, Data-Stream Sampling: Basic Techniques and Results, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 13–44.