Adaptive backpressure algorithm for reactive data flows

KIRILL CHERNYAVSKIY

In a data–intensive network application, the reactive streams specification could be applied for handling data flows, increasing resource consumption of the server machine. It's achieved by integrating backpressure into the data flow model. The benefit of this approach is an increased ability of resources control and easy auto-scaling server configuration. Reactive data processing requires two parts of the data flow: producer and consumer. One end of the flow could be a socket in a common data-driven network application, and another could be a storage layer. There are many different implementations for reactive processing of network connections, such as Netty, VertX, WebFlux, but not so many implementations for working with a file system. All current reactive implementations of file-system consumers and producers are not very good in terms of memory consumption and processing speed in an asynchronous application. In order to fix this gap, the new algorithm for managing backpressure on file-system consumers was introduced and implemented in an open-source library github.com/cqfn/rio providing reactive API for the file-system objects.

1 BACKGROUND

The reactive streams initiative was created to provide a standard of asynchronous data flow processing with non-blocking backpressure. This initiative designed the API for Java and JavaScript, and it was integrated into the standard Java library with JDK9 release as `java.util.concurrent.Flow` package. Reactive Streams describes the problem they are solving as follows:

> The main goal of Reactive Streams is to govern the exchange of stream data across an asynchronous boundary; think passing elements onto another thread or thread pool while ensuring that the receiving side is not forced to buffer arbitrary amounts of data. In other words, backpressure is an integral part of this model in order to allow the queues which mediate between threads to be bounded.

In the reactive-streams manifesto, the main characteristics of the reactive system are described as responsive, resilient, elastic, and message-driven.

The core component of reactive streams is the non-blocking backpressure, meaning that the control of data flow is managed by the consumer (data receiver), which can request the next portion of data items from the producer asynchronously. The producer will send the data when it is ready. Each application can implement a backpressure strategy by itself. The most common strategies are:

- "All At Once" — the consumer requests all data in one request, the producer sends it when ready. It could be used when the consumer can process data much faster than the producer send it, e.g. when calculating the length of file on disk, the `sum` operation will add two numbers always faster than data read from disk.
- "One By One" — the consumer requests the next chunk after receiving a previous one; the producer sends a chunk and waits for the next request. This strategy is usually applied when the producer is much faster than the consumer, e.g. a producer is a software data-generator, and the consumer is a slow file on disk.

Author's address: Kirill Chernyavskiy, g4s8.public@gmail.com.

Building a reactive system requires additional effort from a programmer. However, it gives efficient resource control, which can be helpful in the case of data–intensive applications under high load, such as accurate memory monitoring and automated horizontal scaling.

Java API for reactive streams consists of 4 main components:

- **Publisher** — data producer, an actor who provide the data for `Subscriber`, e.g. reading network socket, reading a file, or data-generator Java class.
- **Subscriber** — data consumer, an actor who receives the data from `Publisher`, e.g. writing the network socket, writing a file, or some reducer, such as hash digest calculator.
- **Subscription** — the channel between `Publisher` and `Subscriber`. `Subscriber` uses this channel to request the next amount of data chunks from `Publisher` or cancel the operation.
- **Processor** — optional intermediate component, which can transform data sent by `Publisher` to `Subscriber`.

2 PROBLEM

Reactive primitives are frequently used to provide interfaces for some data resources, such as sockets, files, HTTP servers and clients. There are multiple implementations for HTTP clients and servers, such as Netty, Spring, VertX. Netty also supports raw socket connection; JDK11 includes a new HTTP client with reactive-streams support; AWS has reactive SDK for S3 storage; VertX has reactive database drivers.

However, there are not enough solutions for file-system resources (i.e. file objects); all existing implementations do not provide expected performance or reliability.

3 RELATED WORKS

There are two more or less popular and stable libraries providing file-system reactive IO:

- **VertX** — VertX mainly provides server components; at the same time, it has additional packages for file-system reactive classes: `AsyncFile`. Since it is not the primary project component, it is not well-documented, and project maintainers don't pay enough attention to this package. It uses the "Request All at Once" backpressure strategy when writing to the file, assuming the file is always faster than any producer.
- **WebFlux** — Spring component that provides multi-purpose reactive SDK. As for VertX, file-system operations are not the primary goal of this SDK. It is more stable than VertX and shows better performance, but it uses the same "Request All at Once" backpressure strategy.

All existing libraries use the same backpressure strategy. It could be suitable if the file Write operations were always faster than any other operation, but it's not the case. This strategy leads to high memory usage for keeping data buffers in the producer until the consumer handles all of them.

4 SOLUTION

Generally speaking, the consumer has an internal queue. It keeps ordered items received from the producer because the producer sends them asynchronously, and the consumer cannot handle all received items immediately. In the case of the "All At Once" backpressure strategy, the consumer requests all items

using a subscription, keeping all received items in a queue to process them one by one. For the "One By One" case consumer may not have a queue since it requests the next item only when it processes the previous one.

To fix the problem described below, we need to invent an algorithm implementing a new backpressure strategy that adopts different consumer throughputs and handles consumer throughput changes.

A new additional component was introduced to define backpressure characteristics: `Greed` — it is responsible to request items from the `Subscription` connection. At the same time, it receives notifications from the consumer about processed and received data items, and it decides how and when to request the subsequent chunks from the connection. The greed keeps two volatile parameters to configure the behavior of the request: `amount` and `shift` — amount represents the amount of data items to request from the subscription, and shift is the sequence shift (in items) for the time of the next request. E.g. if the amount is 100 and the shift value is 2, then the greed will request the next 100 items when it received the notification about the 98th processed item. The greed has initial values for these parameters but may decide to change it later.

In order to be able to make a decision, the greed memorizes the size of the queue (called *queue* later). Greed gets this value by computing the number of "receive" notifications from `Subscriber`. Also, it memorizes the current *position*. When the consumer receives a new item, the greed increments *queue* size and *position*; when an item is processed, the greed decrements the *queue* size. When the consumer notifies the greed about the item received, the greed checks whether the current *position* is eligible for requesting the next amount of items from the subscription connection. It is eligible if:

$$position = amount - shift \tag{1}$$

If the *position* is eligible, the greed requests *amount* of items from subscription and reset current *position* to zero.

Also, the greed is trying to adjust the *amount* and $shift$ variables on each eligible position. First, it is trying to understand whether the consumer's queue processes items faster than greed is requesting by comparing the current *queue* size with the current $shift$ variable: $queue < shift$ means that the consumer is faster than the producer; $queue > shift$ means that consumer is slower than the producer; $queue = shift$ means that producer and consumer are balanced. If the consumer is faster than a producer, the greed is changing *amount* and $shift$ variables:

$$queue < shift \tag{2}$$

$$amount' = amount * 2 \tag{3}$$

$$shift' = min(shift + 1, 3) \tag{4}$$

If the consumer is slower than the producer:

$$queue > shift \tag{5}$$

$$amount' = max(amount/2, 3) \tag{6}$$

$$shift' = max(shift - 1, 1) \tag{7}$$

These constants and formulas were obtained empirically using benchmark tests; there is no background for them yet. It was attempted to run benchmark tests with different constants, and these values seem to be more optimal.

This new backpressure strategy could be called “Adaptive” because it changes self behaviour to be more productive in changing environment. For this “Adaptive” backpressure strategy, the consumer implementation must have an internal queue for received data items, like for the “All at Once” strategy. Also, the consumer implementation should use the greed component to delegate subscription communication instead of managing it by itself. If the queue size is growing (the consumer cannot process incoming requests instantly), the greed is adjusting to request fewer items and requests it when the queue is almost empty. However, when the consumer is fast and processes items faster than the producer delivers, then the greed is adjusting to request more items and beforehand.

5 DELIVERABLES

Using this backpressure algorithm, a new software library was created, providing reactive API for the file-system API: github.com/cqfn/rio. It implements this algorithm in `org.cqfn.rio.WriteGreed` class, which is used by `org.cqfn.rio.channel.WriteTaskQueue` class, which apply write IO operations on the Java channel.

For the reactive-streams specification correctness verification, the technology compatibility kit (TCK) created by the reactive streams organization was used to create specification tests located in `org.cqfn.rio.channel.WritableCh` class.

5.1 Benchmarks

The benchmarks were created to compare the performance of the new library with current VertX and WebFlux implementations. They are located in /benchmarks project sub-directory. These benchmarks are testing three implementations in three different scenarios: read, write and copy files. However, only write tests are related to the new backpressure algorithm. These benchmarks have two parameters:

- item size — the size of data to write (in bytes).
- parallelism level — the amount of parallel write operations.

Benchmarks were run against 3 libraries using the same set of parameters for each of them. The results of the benchmarks are presented in tables 1; it was computed on AWS EC2 m4.large with 40GB SSD io2 20000 IOPS machine.

1Kb files						
Provider	Count	Parallelism	AVG	STDDEV	STDERR	Speed
Rio	1000	1	0.3539	0.7392	0.0234	2.76 MB/s
VertX	1000	1	1.0352	0.8328	0.0263	965.95 KB/s
Flux	1000	1	0.4171	0.5192	0.0164	2.34 MB/s
Rio	1000	10	0.5411	0.6698	0.0212	18.05 MB/s
VertX	1000	10	2.6786	1.4254	0.0451	3.65 MB/s
Flux	1000	10	1.0754	0.7092	0.0224	9.08 MB/s
Rio	100	100	2.2331	1.2984	0.1298	43.73 MB/s

VertX	100	100	12.1945	1.5494	0.1549	8.01 MB/s
Flux	100	100	8.8625	2.8468	0.2847	11.02 MB/s
Rio	100	1000	12.4153	5.2551	0.5255	78.66 MB/s
VertX	100	1000	68.4329	13.8412	1.3841	14.27 MB/s
Flux	100	1000	30.4435	15.6309	1.5631	32.08 MB/s

1Mb files						
Provider	Count	Parallelism	AVG	STDDEV	STDERR	Speed
Rio	1000	1	1.9411	0.8894	0.0281	515.16 MB/s
VertX	1000	1	18.1896	4.3730	0.1383	54.98 MB/s
Flux	1000	1	4.4891	2.9077	0.0920	222.76 MB/s
Rio	1000	10	13.1222	2.6641	0.0842	762.07 MB/s
VertX	1000	10	186.4106	31.6588	1.0011	53.65 MB/s
Flux	1000	10	44.0761	9.5911	0.3033	226.88 MB/s
Rio	100	100	138.6782	11.2211	1.1221	721.09 MB/s
VertX	100	100	1862.1139	9.8860	0.9886	53.70 MB/s
Flux	100	100	487.2991	23.8710	2.3871	205.21 MB/s
Rio	10	100	167.9636	31.4381	9.9416	595.37 MB/s
VertX	10	100	1864.4311	0.3600	0.1138	53.64 MB/s
Flux	10	100	768.8858	127.8018	40.4145	130.06 MB/s
Rio	3	300	483.9835	49.2637	28.4424	619.86 MB/s
VertX	3	300	5949.9329	255.9340	147.7636	50.42 MB/s
Flux	3	300	2375.1227	463.0244	267.3273	126.31 MB/s

10Mb files						
Provider	Count	Parallelism	AVG	STDDEV	STDERR	Speed
Rio	100	1	18.3807	5.7307	0.5731	544.05 MB/s
VertX	100	1	177.7372	26.0655	2.6066	56.26 MB/s
Flux	100	1	49.0719	18.0751	1.8075	203.78 MB/s
Rio	10	10	152.0109	26.1508	8.2696	657.85 MB/s
VertX	10	10	1862.8861	1.1886	0.3759	53.68 MB/s
Flux	10	10	487.6816	63.2540	20.0027	205.05 MB/s
Rio	5	20	310.1501	50.0047	22.3628	644.85 MB/s
VertX	5	20	3720.0799	12.7777	5.7144	53.76 MB/s
Flux	5	20	1149.4646	208.4034	93.2009	173.99 MB/s
Rio	3	30	461.6671	65.1967	37.6413	649.82 MB/s
VertX	3	30	6052.7234	377.2584	217.8102	49.56 MB/s
Flux	3	30	2077.4746	343.0548	198.0628	144.41 MB/s

100Mb files						
Provider	Count	Parallelism	AVG	STDDEV	STDERR	Speed
Rio	50	1	187.2466	7.9968	1.1309	534.06 MB/s
VertX	50	1	1997.0412	581.0320	82.1703	50.07 MB/s
Flux	50	1	431.7884	12.1490	1.7181	231.59 MB/s
Rio	5	10	1447.0634	78.4022	35.0626	691.05 MB/s
VertX	5	10	13318.6614	4689.5994	2097.2526	75.08 MB/s
Flux	5	10	5021.3340	263.5366	117.8572	199.15 MB/s

Rio	3	20	35626.4839	498.7560	287.9569	56.14 MB/s
VertX	3	20	19602.3857	1003.8309	579.5621	102.03 MB/s
Flux	3	20	37166.2301	261.4055	150.9226	53.81 MB/s

Table 1.
Write benchmarks comparing "Rio", "VertX", and "WebFlux" libraries on writing files reactively. The legend: **Provider** – the name of target library under benchmark; **Count** – performed benchmarks count; **Parallelism** – the number of parallel threads performing write operation; **AVG** – the average time for a test run; **STDDEV** – standard deviation for a test run; **STDERR** – standard error for each test run; **Speed** – file write speed in specified units;

6 CONCLUSION AND FUTURE WORK

As we can see from benchmark results, the new algorithm's performance is better than analogues with other backpressure strategies. This library was successfully integrated into the Artipie project as a core component of binary key-value storage. During the implementation process, it becomes clear that this approach is suitable for file-system-based reactive streams and any general reactive data flows with unknown or unpredictable throughput. Therefore, it was redesigned to accept any Java data channel as a data destination.

This algorithm seems to be new for "Queueing Theory"; it resembles the "M/M/I" queue type of "Classical Queueing System" with two additional parameters in the model (amount and shift). It would be interesting to build a mathematical model for this algorithm and find optimal constants currently resulting from empirical analysis.