Effectively-once Semantics in Wallaroo

John Mumm
Head of Architecture
john@wallaroolabs.com

Scott Fritchie
Senior Software Engineer
scott@wallaroolabs.com

Effectively-once Semantics in Wallaroo

Wallaroo is an efficient, low-footprint, event-by-event data processing engine with on-demand scaling. Wallaroo falls into the distributed streaming category along with tools like Apache Flink, Spark Streaming, and Storm. One of our long-running design goals for Wallaroo has been to guarantee effectively-once semantics for use cases where strict correctness guarantees are important, even in the face of failure. In this whitepaper, we'll describe what we mean by effectively-once semantics, explore how Wallaroo is able to make this kind of guarantee, and finally describe the testing strategies we've employed to verify these claims.

I. What do we mean by "effectively-once semantics"?

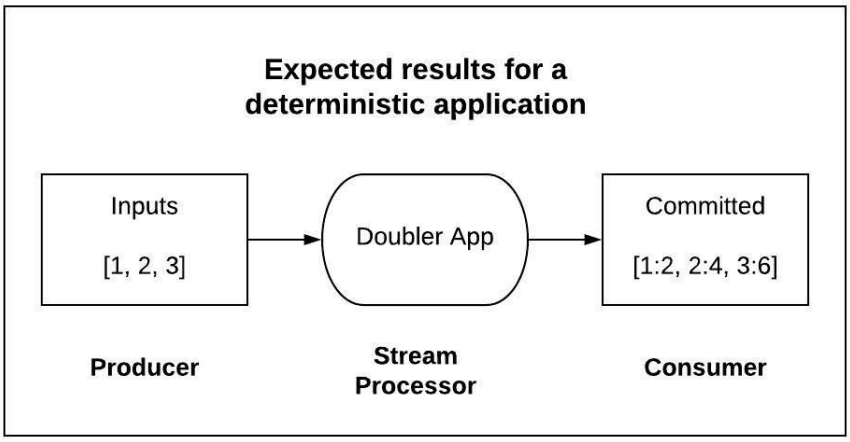
Stream processing systems make certain kinds of guarantees around message processing. We can frame these guarantees in terms of the relationship between the set of inputs sent into the system and the set of outputs produced by the system. In the ideal case, a set of inputs will correspond to a set of valid outputs. For deterministic applications, there will be one set of valid outputs. For nondeterministic applications, there can be more than one set of valid outputs, since different executions can lead to different results. In what follows, we assume that any internal application logic is bug-free, meaning that, if a subset of inputs is successfully processed by a computation, that computation will always generate a valid subset of outputs.

When talking about stream processing semantics, we're particularly interested in the point at which the outputs corresponding to a given input are "committed." If an output is committed, then its effects are observable outside the closed system internal to Wallaroo. For example, outputs can lead to updates to a database, or they can be written to a file or Kafka topic. Stream processing semantics involve guarantees around the following properties:

1) Do the committed outputs leave out expected data? In this case, messages were lost.
2) Do the committed outputs include duplicates?
3) Do the committed outputs include incorrect data (even when the application logic itself is bug-free)?

With these properties in mind, we can distinguish five main categories of stream processing semantics: zero-guarantee, correct-if-committed, at-most-once, at-least-once, and effectively-once. Let's explore the differences between these.

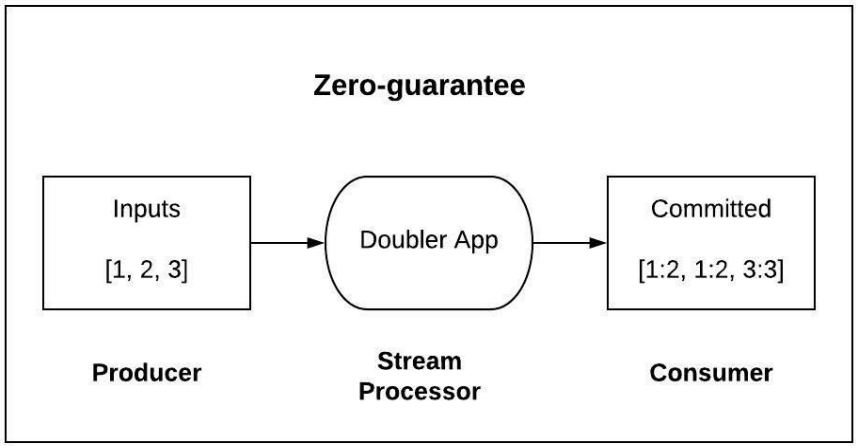
To begin with, ignoring questions about failure and distributed systems for the moment, imagine we have a simple doubler application that assigns exactly one output to each input integer. If we send inputs into this system, we expect to see one and only one output committed per input:



This is the ideal case, and what one would like to guarantee even in the presence of failure. The outputs include both the input and the transformed value (e.g. "1:2") as a shorthand for the fact that we need to correlate inputs to outputs in order to reason about our processing guarantees. We can understand different kinds of processing semantics by contrasting them with this ideal case in terms of "what can go wrong."

Zero-guarantee

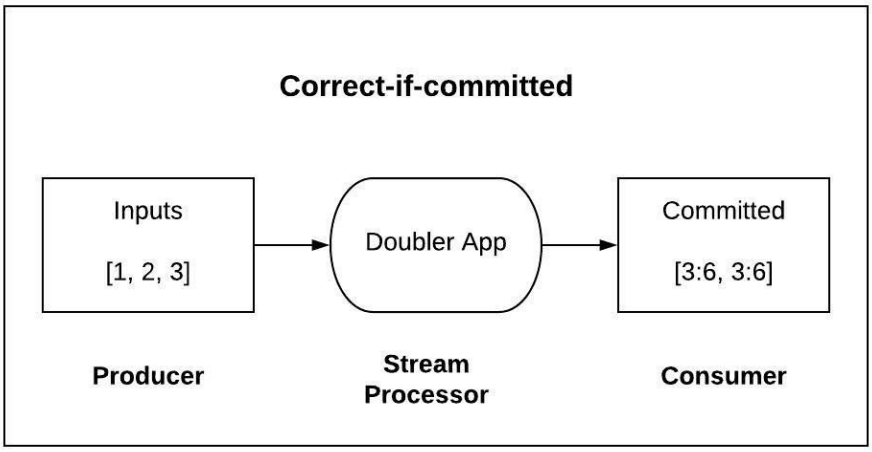
In the case of *zero-guarantee semantics*, given an application and a set of inputs, we can't say anything ahead of time about which outputs will be committed. It's possible that the set of outputs will be valid. But it's also possible that some of the outputs will be duplicates. Furthermore, it's possible some of our outputs will have been lost. In fact, we don't even know if the committed outputs will be correct (even with bug-free application logic). Here's a possible run under these (effectively nonexistent) guarantees:



In this example, we commit a correct answer for input *1*, but we commit it twice. We commit no answer for *2* (it was apparently lost), and we get an incorrect answer for *3.* This was just one possible run. It's also possible to get fewer than 3 outputs or more than 3 outputs under these conditions. A system like this is untrustworthy, to say the least.

Correct-if-committed

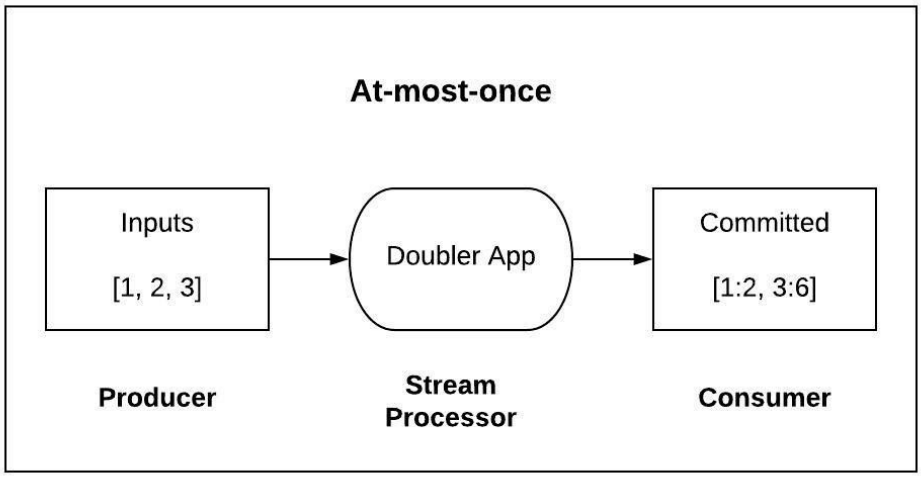
A bare minimum for a useful system is the guarantee that all committed outputs are correct. Even with bug-free application logic, there are a number of ways things can go wrong. Data can be corrupted over the network, or by the framework itself. Or perhaps the framework is buggy and routes messages to the wrong computations. Whatever the possible cause, a system must rule out this kind of corruption through extensive testing. Correct-if-committed makes no guarantees about lost or duplicated messages, but it does say that only correct outputs will ever be committed. Here is a possible run under this guarantee:



We've lost the outputs for *1* and *2*, and we've committed the output for *3* twice. But because we committed it, given this guarantee, at least we know that the doubler app's calculations are correct, e.g., 3 * 2 is indeed 6.[1] For the remaining three categories of processing semantics, we will assume that the *correct-if-committed* guarantee holds.

At-most-once

Perhaps for your application, if outputs are guaranteed to be correct, it's acceptable to lose messages sometimes, but it's crucial that the output set contains no duplicate messages. In this case, you'd prefer *at-most-once semantics,* which guarantees that no output will ever end up being committed twice. From the viewpoint of any external system, each input will have been processed at most once. Fire-and-forget could achieve this goal, as long as we are able to prevent the same input from being fired twice. Here's a possible run under this guarantee:

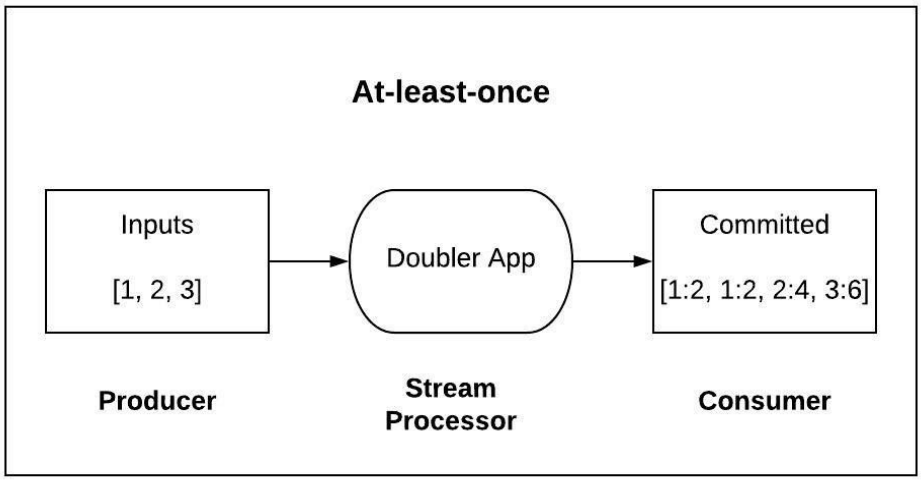


We have committed the correct outputs for *1* and *3*, but the output for *2* appears to have been lost. The important point is that we have committed no duplicate outputs.

[1] In general, not all inputs must have corresponding valid outputs. For example, if the app logic filters out a certain class of inputs, then we should not expect outputs corresponding to them in our set of committed outputs. In this doubler example, we expect every input to have a corresponding output, since it's a simple one-to-one function. Since correct-if-committed makes no guarantees regarding lost outputs, however, it only tells us that any outputs committed will be in the set of valid outputs. Some outputs might be missing, as in the diagram.

At-least-once

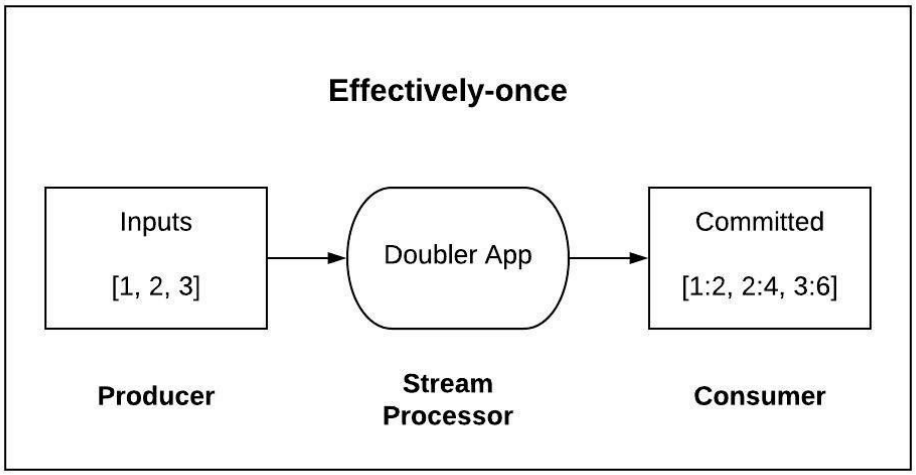
Perhaps you can tolerate duplicates, which are ruled out by at-most-once semantics, but you can't tolerate lost messages. In this case, you'd prefer *at-least-once semantics*, which guarantees that at least one of each valid output will be committed (but says nothing about how many duplicates of each will show up). Under this guarantee we could see something like this:



There are no lost outputs, and all the results are correct. But because we're only guaranteeing *at-least-once*, in this case we've ended up committing duplicate outputs for *1*.

Effectively-once

If you want to rule out both lost messages and duplicates, then you need *effectively-once semantics*. Effectively-once guarantees that for a given application and set of inputs, the set of outputs committed will be exactly equivalent to some set of valid outputs. This output set will include all expected data, and will include no duplicate data. We use the term "effectively-once" rather than the more familiar "exactly-once" because it is clearer that we do not mean to say, for example, that every output is literally sent over TCP exactly once. The guarantee is that from the viewpoint of any external system, it will look as though each input was processed exactly one time. So with our deterministic Doubler application, and with the inputs *1, 2,* and *3*, we guarantee the following result:



Every input has a corresponding (correct) output. And no output has been committed more than once.

II. How does Wallaroo guarantee effectively-once semantics?

Wallaroo supports end-to-end effectively-once semantics, provided that producers are replayable and consumers can participate in checkpoint-based transactions with Wallaroo sinks (or, in the case of a certain class of deterministic applications, if consumers are idempotent). Even in the absence of these guarantees at the producers and consumers, Wallaroo provides consistent distributed snapshots taken at tunable intervals that can be used to roll back in the presence of failures.

In order to achieve effectively-once semantics, we need to guarantee no committed outputs are lost or duplicated, and that all committed outputs are valid outputs. In a completely deterministic application, "all outputs are valid outputs" means that the one and only correct set of outputs is committed by a consumer. However, there are many ways that nondeterminism can enter into Wallaroo applications. Here are some examples:

- Because of parallelism and Wallaroo's key-based partitioning scheme, Wallaroo only guarantees ordering by routing key and not across keys. If application operations are not commutative, this can lead to different outputs on different runs of the same inputs (e.g. if the final operation is appending to a list).
- If the operations themselves are not deterministic (e.g. they use clock time or randomization), then different runs with the same inputs can lead to different outputs.

- Depending on how window policies are set via the data windowing API, receiving the same inputs at different times can lead to different outputs. For example, just because an input is received before a configured late data threshold on one run does not mean it won't be considered late on another run where the relative timing of inputs is different.

There are other examples, but the point is that we must account for possible nondeterminism from run to run, given the same inputs. This is why "correct" is defined as an outcome in which "all outputs from Wallaroo are valid outputs given the set of input messages". Think of "valid outputs" as the set of valid sets of outputs (each one corresponding to a possible valid execution).

In the face of failure, there are three core challenges that we must overcome in order to guarantee effectively-once semantics:

1) Ensure we can replay any messages from a producer that were lost so that the corresponding outputs can be committed at the consumer.
2) Ensure that any outputs we already committed do not end up getting committed at the consumer again when we replay.
3) Ensure that we can recover any intermediate state that will not be reconstructed by replaying lost messages.

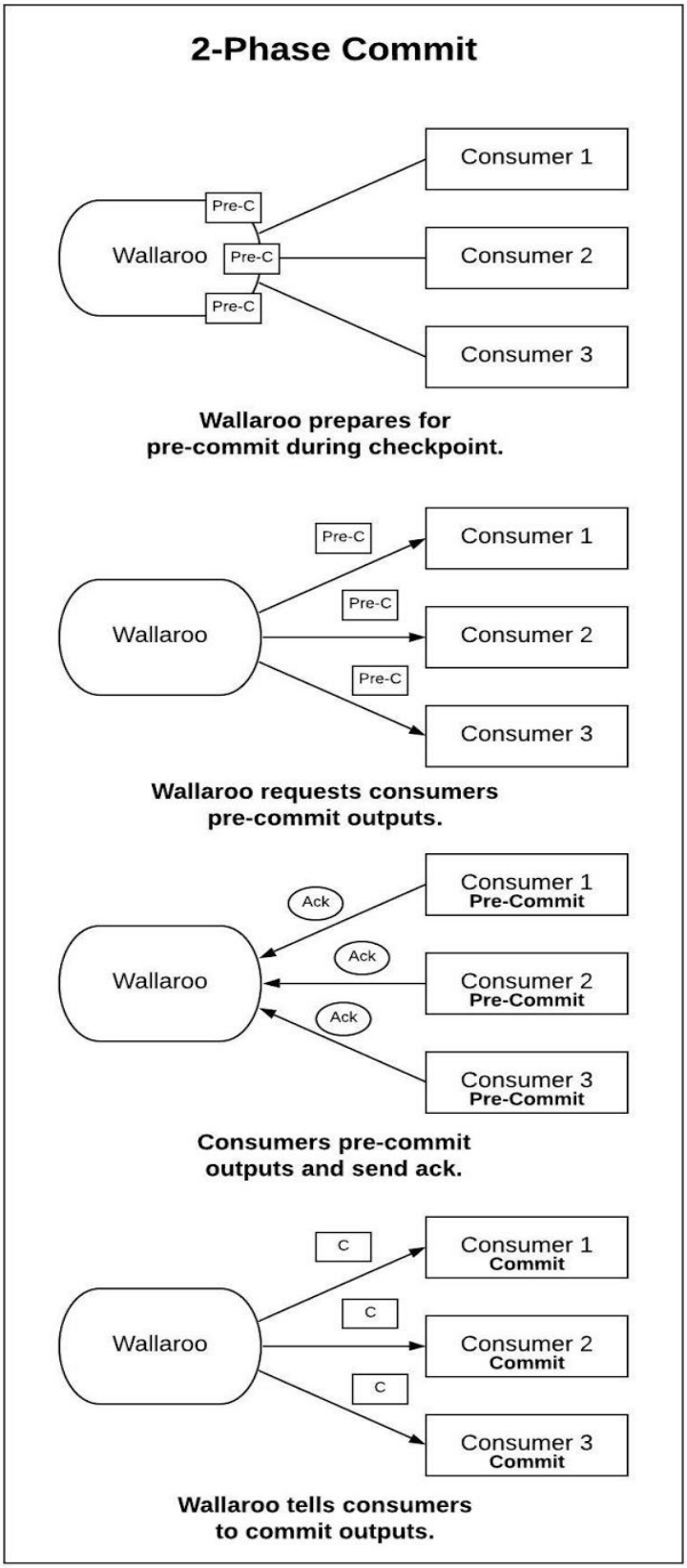
To achieve these three goals, we use a checkpointing protocol to periodically take global snapshots of Wallaroo state, including metadata about which inputs we have already seen and which outputs we have committed to a consumer. Our protocol is based on the Chandy-Lamport algorithm and some of the modifications proposed in "Lightweight Asynchronous Snapshots for Distributed Dataflows". We chose our protocol because it ensures consistent checkpoints while adding relatively little overhead, and without the need to stop processing globally.

On recovery from failure, Wallaroo rolls back to the last successful checkpoint. The system can then request replay from producers when necessary. For any checkpoint, we partition all messages processed into those that happened before the checkpoint and those that happened after the checkpoint. This means that for each producer, we know the last input received before the checkpoint, and can request that the producer starts replaying at the message immediately after that one. Wallaroo will also drop any inputs that it has already successfully processed to avoid duplicate processing.

Notice that we are assuming the producer has the capability of replaying from a given point in the input stream. If the producer does not have this capability, then we can't even guarantee *at-least-once* semantics. For example, perhaps the producer believed it sent a message, but Wallaroo crashed before receiving it. No matter what Wallaroo does at this point, if the producer can't replay the message, we've lost access to that message.

Things are even trickier on the consumer side. We must guarantee that all outputs are committed at the consumer, and that they are committed only once. But we must also guarantee that the set of outputs committed is valid. Notice that idempotence at the consumer (without further qualifications) is not sufficient for effectively-once from that consumer's perspective. That's because, in the presence of nondeterminism, the same checkpoint subset of inputs can lead to a different set of outputs. Whenever we roll back to a checkpoint, we replay all messages starting from where that checkpoint left off. Imagine we roll back to the same checkpoint 3 times in the course of a run. Each time, let's say we replay only one message successfully before crashing again. In the presence of nondeterminism, we might produce a different number of outputs corresponding to that input each time. Even with idempotence at the consumer, it might end up committing all of these outputs, which might not be a possible result of a *single* valid execution. Hence, we fail in this case to guarantee effectively-once processing. This is only one example of how things can go wrong if you rely solely on idempotence in the presence of nondeterminism.

Because of challenges like these on the consumer side, we use a 2-Phase Commit (2PC) protocol for consumers. This allows us to ensure that all (and only) outputs associated with a successful checkpoint are committed. In its first phase, this protocol waits for all consumers to durably pre-commit messages related to a checkpoint. In the second phase, if every consumer has successfully pre-committed, we tell every consumer to commit. Otherwise, we abort the checkpoint. Wallaroo provides a producer/consumer protocol that allows different consumers to implement pre-commit and commit in different ways. A message should only be "processed" by a consumer if it has been committed according to 2PC. Here is a diagram showing the steps of a successful 2PC:



Every checkpoint triggers 2PC, which involves all consumers participating in the checkpoint, and the checkpoint is only successful if the 2PC succeeds. There are two crucial properties ensured by 2PC:

1) A consumer will never commit outputs unless they are associated with a *successful* checkpoint (meaning that we could roll back to that checkpoint, and that the inputs associated with that checkpoint will never be replayed). [Safety property]
2) All outputs associated with a successful checkpoint will *eventually* be committed. [Liveness property]

If we are going to use checkpoints to support effectively-once semantics, we need to make sure that the checkpoints themselves are stored in a resilient fashion. After all, machines containing those checkpoints could die and their file systems could be lost. In order to address this problem, we need to use some form of data replication. When Wallaroo is run in data replication mode, all file-altering I/O to resilience and checkpoint-related files is written to journals, or write-ahead logs. The journal files are replicated to a remote file server, to protect against catastrophic worker failure.

It was an explicit goal to limit any Wallaroo journal replication service to as few features as possible. For example, the ability to do random write I/O can violate safety: overwriting part of a journal can make Wallaroo roll back to impossible or incorrect state. A limited-capability file server can stop bad behavior by a buggy client by simply not being able to do a bad thing. Wallaroo's remote file server is called "SOS", the Simple Object Service. It is quite similar to the FTP protocol, except with even fewer operations and with a binary protocol instead of ASCII-based protocol.

For more on our checkpointing protocol and consistent recovery lines, see https://blog.wallaroolabs.com/2018/10/checkpointing-and-consistent-recovery-lines-how-we-handle-failure-in-wallaroo/. For more on our data replication strategy, see https://blog.wallaroolabs.com/2018/11/the-treacherous-tangle-of-redundant-data-resilience-for-wallaroo/.

III. How do we validate these guarantees?

Prior sections have described the effectively-once semantics that Wallaroo is designed to provide and the mechanisms used to implement them (such as checkpointing and state replication). This section describes how we test that Wallaroo's behavior meets effectively-once criteria.

We first start by describing how we would test a hypothetical effectively-once system built upon the UNIX utility cat(1). Then we briefly discuss the limitation of cat(1) with respect to

effectively-once criteria, and then extend the analogous cat(1)-based system with UNIX shell pipelines. The method for injecting faults into the UNIX shell pipeline and verifying the pipeline's output are very similar to what we actually do to test Wallaroo's behavior for effectively-once properties.

Effectively-once testing by analogy: testing cat(1)

Let's try to design a test for a UNIX utility, cat(1)[2], for a hypothetical effectively-once data processing system. We want to see if cat can violate properties of effectively-once data systems:

a. No duplicated data in its output
b. No lost/dropped data in its output
c. No corrupted data in its output

The cat utility's basic use is quite simple:

1. Try to read some block-size number of bytes from the stdin file descriptor. The input file is divided into small records, e.g. 50 bytes, and each record is unique.
2. If any bytes were read, then write those bytes as-is to the stdout file descriptor, then goto step 1.
3. If no bytes were read, then the end of input has been reached: close all file descriptors and exit.

If we run a simple test case like this at a UNIX login shell:

```
$ cat < input-file > output-file
```

and then compare the contents of the two files:

```
$ cmp input-file output-file
```

the cmp(1)[3] utility will probably tell us (via an exit status of zero) that the two files are identical, and thus we know that cat did not not corrupt the output: no duplicated, dropped, or corrupted data was written to the output file.

In our next test of cat, we will kill the cat process in the middle of its processing. Let's assume that our fingers are quick and the input file is very big, so that we can kill the process by pressing Control-c before the process would exit normally.

[2] We will refer to the cat(1) program as "cat" from now on.
[3] We will refer to the cmp(1) program as "cmp" from now on.

```
$ cat < input-file > output-file
<Press Control-c>
$ cmp input-file output-file
cmp: EOF on output-file
```

This time, the files are different. The output file is smaller than the input file. It makes sense: we killed the cat process with a SIGTERM via Control-c before it had finished its work. However, cmp is also implicitly telling us that all of the bytes in the output file are identical to the input file. That's good: no dropped, duplicated, or corrupted data.

What if we were to restart the cat process? cat is not intended to provide any of the three effectively-once behaviors that we desire: no dropped, duplicated, or corrupted data. But, let's try it anyway.

```
$ rm -f output-file
$ cat < input-file >> output-file
<Press Control-c>
$ cat < input-file >> output-file
$ cmp input-file output-file
cmp: input-file output-file differ: char 13091, line 247
```

cmp tells us that the files aren't identical. It turns out that offset 13,091 in the output files contains duplicate data, i.e., from the beginning of the input file. This shouldn't surprise us: the second cat process didn't know to start reading from anywhere other than the beginning of the input file.

For our next test, let's try to build a pipeline of cat processes and kill one of them.

```
$ rm -f output-file
$ cat < input-file | cat | cat >> output-file
<Kill the 2nd cat process>
Terminated
$ cat < input-file | cat | cat >> output-file
$ cmp input-file output-file
cmp: input-file output-file differ: char 38160, line 720
```

This also shouldn't surprise us: the first cat process in the second pipeline didn't know to start reading from anywhere other than the beginning of the input file.

When our analogy starts to fail

Our hypothetical system using cat is starting to show its limits. The limits include:

a. A simple Bourne/Bash login shell cannot restart a single process within a pipeline--but it would be really useful if it could. Then we could kill any process in the pipeline, restart that process, and then observe what happens to the output file.
b. A cat process cannot communicate or synchronize with other cat processes except via the data it writes to its stdout file descriptor.
c. The UNIX shell pipelines described are strictly uni-directional: it is impossible for cat processes to communicate with other cat processes upstream of them. (We'll ignore the propagation of UNIX signals: they don't carry enough information to be useful to us.)

Wallaroo's processing pipelines are similar to UNIX pipelines

If we straighten out our hypothetical cat pipeline from its original:

```
$ cat < input-file | cat | cat >> output-file
```

to this form (with invalid shell syntax):

```
$ input-file > cat | cat | cat >> output-file
```

then we aren't too far away from a simple Wallaroo system that utilizes:

a. A producer (a.k.a. "connector source" in Wallaroo's docs & source code) to convert byte-oriented data to a common message-oriented format that Wallaroo can easily parse and use (called producer in the Diagram 1 below). The producer reads ASCII text files with newline separators and sends the data to Wallaroo, one text line per Wallaroo input message.
b. A simple Wallaroo application called "passthrough" that behaves like cat does: read message from an input source and write them as-is to a consumer process.
c. A consumer (a.k.a. "connector sink" in Wallaroo's docs & source code) back-end to convert the message-oriented Wallaroo sink output into a byte-oriented stream of bytes (called consumer in the Diagram 1 below).
d. Communication in and out of Wallaroo is done over TCP instead of over UNIX pipes.

Now we have a system that looks very similar to the multi-cat pipeline:

```
input-file > producer <-TCP-> passthrough <-TCP-> consumer >> output-file
```

Diagram 1: One producer TCP connection, one Wallaroo process, and one consumer TCP connection

Multiple processes may work together as a single Wallaroo application cluster; let's number the Wallaroo processes from 1 to N. On the consumer side, there are K consumer TCP

connections, at least one per Wallaroo process, i.e., K >= N. On the producer side, there can be a far larger number M of producers and/or producer TCP connections, e.g., hundreds or thousands per Wallaroo process. These relationships are shown in Diagram 2.

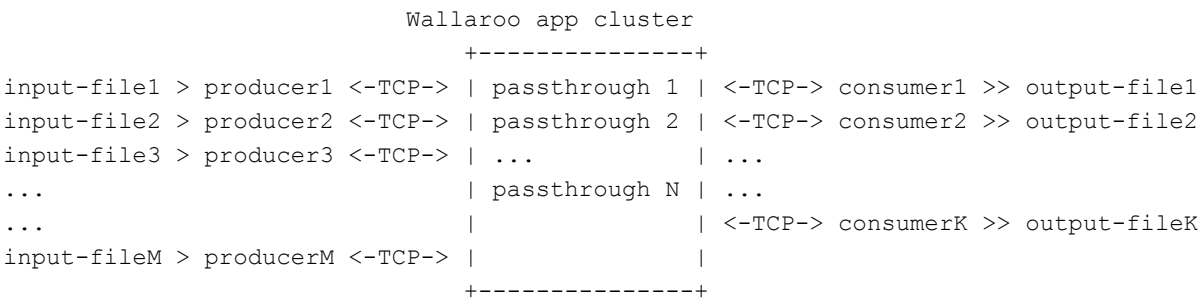


Diagram 2: M producer TCP connections, N Wallaroo processes, and K consumer TCP connections

Unlike a UNIX shell pipeline, Wallaroo processes can communicate bi-directionally with both the producers and consumers via TCP. If there's a failure of a producer/consumer/Wallaroo process, we have more flexibility for a replacement process to coordinate how to resume effectively-once data processing.

When any component in the pipeline fails (producer, passthrough, and/or consumer), Wallaroo's checkpoint rollback mechanism may make some demands on the producer and consumer that aren't possible in the cat scenario:

a. Request that the producer rewind to an earlier offset in the input stream so that data can be re-sent to Wallaroo for reprocessing. This request is performed via the stream offset management built in to the Wallaroo <-> producer protocol.
b. Request that the consumer discard data that had been received prior to the failure. This request is performed via a 2-Phase Commit protocol that is piggybacked in the Wallaroo <-> consumer protocol.

Types of failures and system change injected into the Wallaroo system

In the cat example, we used a single cause of failure to test the system: SIGTERM to kill a cat process. Wallaroo Labs' test procedure for Wallaroo's effectively-once behavior includes the following failures and system change events:

1. Crash and restart any/several/all Wallaroo process (via SIGKILL).
2. Crash and restart the producer process.
3. Crash and restart the consumer process.

Wallaroo output checking during failure & system changes

Recall that the Wallaroo app passthrough is identical to the cat utility in that it copies its input as-is to its output. Therefore, we can use a procedure similar to the cmp utility to verify that the output of the consumer process is a strict prefix of the input file. We also use the "small, unique records" technique for the input file to greatly increase the odds that we can detect output data that has been lost, duplicated, or corrupted.

There is a small complication: the output file created by the consumer can contain tentative (or uncommitted) data from Wallaroo; Wallaroo may not yet have issued a 2-Phase Commit round to confirm that the Wallaroo output is final. Therefore, we must somehow filter out of the output file any uncommitted data.

This filtering is done by a small Python program that understands the format of the transaction log that is maintained by the consumer process. The Python program reads the transaction log, identifies all committed 2PC transactions, and then copies the byte ranges from each committed transaction into a new, filtered output file. We then (literally) use the cmp utility to compare the filtered output file to the input file[4]. If the results are identical, then we are assured that Wallaroo's output has not duplicated, dropped, or corrupted any data.

When Wallaroo is bug-free and correct, what would we see and know?

While the Wallaroo app is running and consuming input and producing output, the following is being controlled by a test harness:

a. Wallaroo's output is constantly being checked via the filter-then-compare-via-cmp technique described above.
b. Wallaroo processes, producers, and consumers are being crashed and restarted. The rate of crashes & restarts is approximately 500-2,000 crash+restart events per hour.
c. Checks that each Wallaroo process is running and has not crashed for a reason other than that a SIGKILL signal was sent by the harness.

The randomness in each part of the test harness creates novel event interleaving that is difficult to achieve using other techniques. We have found some really tough bugs that we had never seen before in old code. We use the same random stress test to gain confidence that those bugs have been fixed. We aren't 100% sure that the bugs are fixed, but we can

[4] We also add the `-n` flag to limit how much of the input file is read.

use randomness to help create similar conditions and verify that Wallaroo's output remains correct.

If the filter-then-compare-via-cmp technique runs without error, then we know:

a. Wallaroo's output is identical to a strict prefix of the input file.
b. Therefore, the output has no dropped, duplicated, or corrupted data.
c. Therefore, Wallaroo output demonstrates effectively-once processing.

When this testing first began, we would find bugs (either incorrect output or Wallaroo internal assertion check crashes) within a few minutes. If the test halted because of a Wallaroo internal assertion failure, we investigate the trace logs of each component to identify if the error was caused by an incorrect assert statement, test harness bug, or genuine Wallaroo bug. All types of bugs are analyzed and fixed, and then the tests resume.

If the cmp check fails, we have encountered a genuine failure of effectively-once processing semantics. These failures are very rare and typically require tests that execute for 10+ hours and are triggering 5-20+K crash+restart cycles successfully before an error is found.

Our testing doesn't prove that Wallaroo is perfect, but it does demonstrate that if Wallaroo violates effectively-once processing semantics it is only very rarely. We are continually testing and addressing additional edge cases to improve Wallaroo further.

IV. Conclusion

One of our long-running design goals for Wallaroo has been to guarantee effectively-once semantics. This means that, even in the face of failure, we guarantee that no messages will be lost, duplicated, or corrupted if all producers are replayable and all consumers can participate in our 2-Phase Commit transaction protocol. We support resilience through a lightweight, asynchronous checkpointing protocol and a journal-based replication scheme. We've subjected our system to a variety of rigorous tests, checking that under a variety of crash scenarios, no expected outputs are lost, duplicated, or incorrect.

Further Reading

John Mumm - Checkpointing and Consistent Recovery Lines: How We Handle Failure in Wallaroo

Scott Fritchie - The Treacherous Tangle of Redundant Data: Resilience for Wallaroo

Paris Carbone, et al. - Lightweight Asynchronous Snapshots for Distributed Dataflows

K. Mani Chandy and Leslie Lamport - Distributed Snapshots: Determining Global States of Distributed Systems

E.N. Elnozahy, et al. - A Survey of Rollback-Recovery Protocols in Message-Passing Systems

Jay Kreps - Exactly-once Support in Apache Kafka

Appendix: Limitations on checking Wallaroo's output correctness

Most Wallaroo applications use a routing key (described in Section II). The routing key in a message is used to route that message to the Wallaroo worker process responsible for computations on messages with that key. Within a Wallaroo worker, the routing key is also used to route the message to a unique actor that performs computations on messages with that key. Wallaroo does not make any ordering guarantees for messages received with different routing keys.

As shown in Diagram 2 in Section III, Wallaroo may have multiple producer TCP connections that feed into a Wallaroo cluster. Wallaroo does not make any ordering guarantees between messages received via different producer TCP connections.

Also in Diagram 2, we see that a Wallaroo cluster sends output to consumers via one or more TCP connections per Wallaroo process. For these consumer TCP connections:

1. The choice of which TCP connection to use is determined by the message's routing key. As long as the routing key remains constant[5], all messages with the same routing key will exit the Wallaroo cluster via a single Wallaroo process and single producer TCP connection.
2. Wallaroo does not make any ordering guarantees for messages sent by different Wallaroo processes.
3. For a Wallaroo process that use more than 1 TCP connection to a consumer, Wallaroo does not make any ordering guarantees for messages sent to multiple consumers.

[5] Routing keys are defined by user-specified functions. Typical Wallaroo use cases do not change or redefine a message's routing key in the middle of a processing pipeline.