



COVR: Composable Output-Valid Resilient Messaging

Alan H. Karp, Marc Stiegler, Terence Kelly

HP Laboratories
HPL-2014-14

Keyword(s):

Resilience; messaging protocol; fault tolerance

Abstract:

As distributed systems scale to larger sizes, the impact of crash-restart failures increases. These failures must be dealt with in a principled manner since they may result in data inconsistencies. Current solutions often prove unworkable at large scale or when the applications are distributed across enterprise boundaries. COVR avoids data inconsistencies with less coordination than other approaches, making it more suitable for modern distributed systems. More precisely, crash-restart failures and dropped or retried messages cannot result in data inconsistency among components that communicate with a COVR-compliant protocol, even if those components were independently written. This paper presents the least constraining set of rules that components must obey to be COVR-compliant, which should prove useful to designers and implementers of distributed systems and messaging infrastructures.

COVR: Composable Output-Valid Resilient Messaging

Alan H. Karp¹, Marc Stiegler¹, and Terence Kelly²

HP Enterprise Services¹, Hewlett-Packard Laboratories²

alan.karp@hp.com, marc.d.stiegler@hp.com, terence.p.kelly@hp.com

Abstract

As distributed systems scale to larger sizes, the impact of crash-restart failures increases. These failures must be dealt with in a principled manner since they may result in data inconsistencies. Current solutions often prove unworkable at large scale or when the applications are distributed across enterprise boundaries. COVR avoids data inconsistencies with less coordination than other approaches, making it more suitable for modern distributed systems. More precisely, crash-restart failures and dropped or retried messages cannot result in data inconsistency among components that communicate with a COVR-compliant protocol, even if those components were independently written. This paper presents the least constraining set of rules that components must obey to be COVR-compliant, which should prove useful to designers and implementers of distributed systems and messaging infrastructures.

1. Introduction

Crashes happen. When a component that communicated with another component crashes and recovers, the result can be sending and receiving components in inconsistent states. Consider a familiar example. Alice uses an app to pay a bill which results in a message being sent to her bank. Alice's app then crashes. When it recovers, it is in the state it was in before the message was sent. That data inconsistency could result in Alice paying her bill a second time and even overdrawing her account. Alternatively, the bank computer could crash after receiving the message but before processing it. In this case, the data inconsistency results in Alice believing that the bill has been paid when it was not. Banking systems go to great lengths to avoid such problems, but the mechanisms they use don't work in general, particularly when governance does not tightly control the components.

The standard approaches deal with data inconsistencies by either avoiding them with distributed transactions or correcting for them with compensating messages. Distributed transactions add significant complexity and require tight coupling among components. Compensating messages don't prevent the bad effects of data inconsistencies and may never catch up with inconsistencies as they propagate through a large scale system. COVR (Composable Output-Valid Resilient) messaging guarantees that crash-restart failures and dropped messages do not induce data inconsistencies among COVR-compliant components. Further, a COVR-compliant protocol can be implemented with decisions based solely on local information, which is particularly significant for large scale systems.

Output validity is a concept that recognizes that interactions among components can be non-deterministic due solely to differences in message arrival order on different runs. Hence, there may be many valid sets of outputs for a given set of inputs even with fully deterministic components. A crash-restart is likely to result in different message arrival order for at least some components compared to a run with no crashes. However, that same message order could have occurred had the computation or message delivery merely been slow. In an output-valid system, the set of outputs from a run in which a node crashed and restarted is indistinguishable from some run with differences in the speed at which the nodes ran and/or the speed at which messages were delivered.

In general, compensating messages and distributed transactions with two-phase commit cannot enforce output-validity. A compensating message might arrive too late to prevent an erroneous output, while there is a failure mode that leaves two-phase commit in an indeterminate state [1] that can lead to a violation of output validity.

Composability is a different matter. Consider two, independently written, output-valid systems. These two systems are composable if they remain output-valid when they start exchanging messages. Systems that interact with a COVR-compliant protocol are composable in this sense.

In what follows, we'll discuss prior work on this problem, present the minimum set of rules for COVR compliance, and provide implementation guidelines.

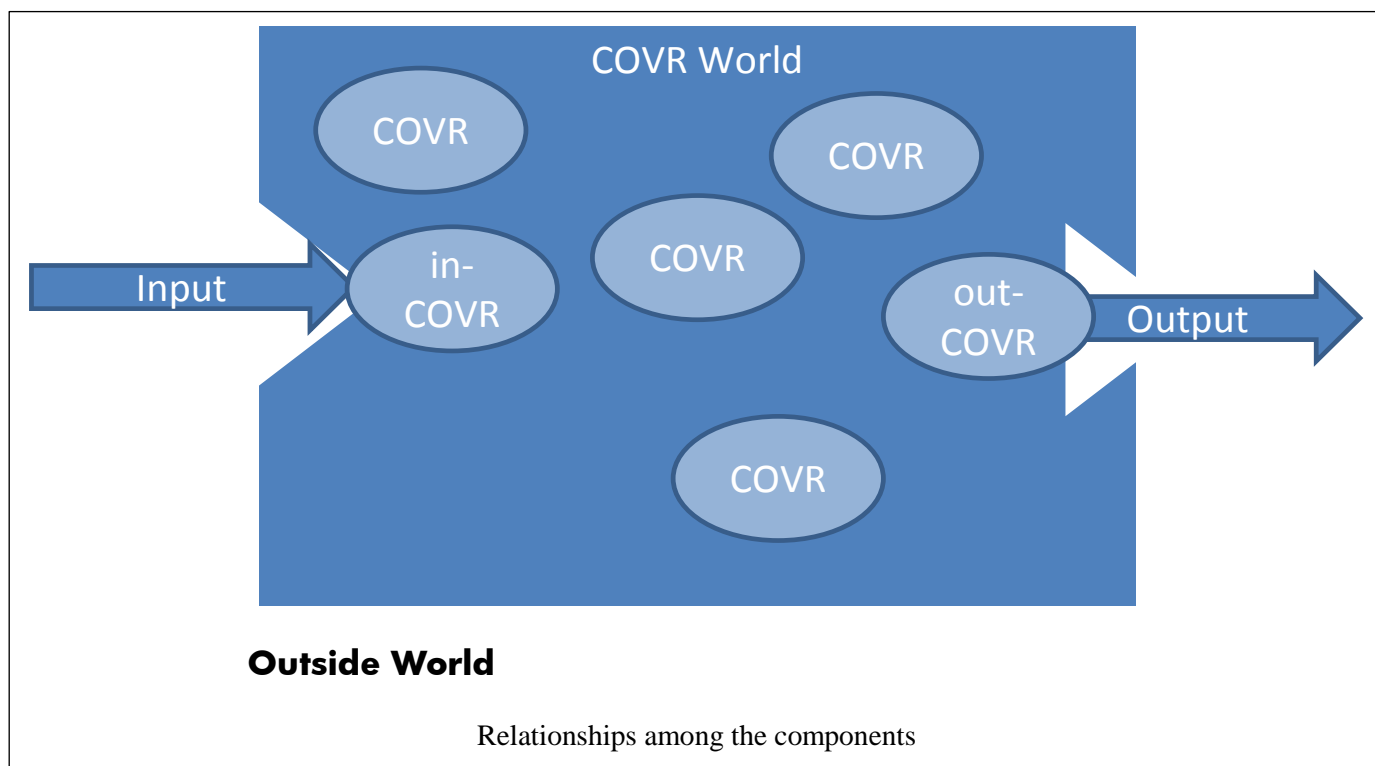
2. Related Work

Waterken [2] provided the first implementation of COVR messaging for distributed programs written in Java and some other languages that run on the Java Virtual Machine. However, waterken was designed to support writing applications that require secure cooperation. The result was COVR features interlaced with security features. For example, certain serializations are not supported because they could lead to exploitable vulnerabilities. A later version, CKen for C/C++ programs [3], separated the resilient messaging from the security aspects of waterken. CKen assumed a number of specific details of the implementation, such as the use of checkpoints for rollback-recovery. In this note, we wish to abstract away all implementation details and present only the minimum set of rules that gives the COVR properties. Relevant non-COVR work is discussed elsewhere [3].

3. COVR Terminology

The essence of COVR is for a component never to send a message it might later need to recall, but there are additional considerations. For example, we also need to ensure that every return value eventually reaches its destination. In order to meet this and other requirements, we assume that every crashed component eventually restarts, and every message or one of its retries eventually reaches its intended destination.

The figure shows the relationships of the components. We divide the world into an inside, *i.e.*, the COVR world, and everything else, *i.e.*, the outside world. Any component that obeys the COVR rules is part of the COVR world, no matter its physical or logical location with respect to other COVR components. Interaction with the outside world is via special components that implement a COVR-compliant protocol when communicating with COVR components and different protocols when communicating with the outside world.



We start the description of COVR with some definitions.

Component: A compute element capable of modifying only its own state and sending messages, return values or outputs to other components, often in response to messages or inputs it receives

Processing: Changing state and/or releasing messages, return values, or outputs in response to a message

Processed: A message that has its effects on the processing component in a state that can survive a crash

Message: Data the sending component wants the receiving component to process

Return Value: Data released to the sender of a message by the recipient

Outside World: Components that do not obey the COVR rules

Input: Data received by a component in the inside world from a component in the outside world

Output: Data sent from a component in the inside world to a component in the outside world

Release: Put a message, return value or output in a state where the sender cannot recall it

Retry: Re-release of a message, considered to be the same message

COVR Component: A component that obeys the COVR rules

COVR'd crash: The crash of a COVR component

in-COVR Component: A COVR component that has a piece in the outside world that receives inputs

out-COVR Component: A COVR component that has a piece in the outside world that produces outputs

The in-COVR and out-COVR components have one foot in the COVR world and one foot in the outside world. The outside world part contains a “driver” specific to the outside world component it is communicating with. The driver is responsible for handling such things as replayed inputs and output acknowledgements, if any. A driver is also responsible for deciding whether an input or output should be at-least-once or at-most-once.

The result of crashes of in-COVR components can be lost inputs if they are not replayed by the sending component. The target of an output can see a lost, partial, or duplicate output should the sending out-COVR component crash. Since the drivers are part of the outside world, exposing the failure of a driver to the outside world when its in-COVR or out-COVR component crashes does not result in any output that could not have been produced in a run with no COVR'd crashes. In other words, such a crash does not violate output-validity.

4. COVR Rules

We'll list the rules first and then explain them. A COVR component must

1. restart from a crash in a state consistent with all messages, return values and outputs it released before the crash;
2. ensure that each message it releases can eventually reach the intended recipient at least once;
3. make each received message or retry of that message available for processing only until one of them has been processed;
4. send the same return value sent the first time a message was processed every time a retry of that message is received.
5. process each input no more than once;
6. make at least one attempt to release each output.

Any component that follows these six rules is COVR-compliant. Any set of COVR-compliant components has the composable output-valid property.

The rules are quite abstract, leaving room for many different implementations. Here we provide some options.

Rule 1: restart from a crash in a state consistent with everything released before the crash

This rule can be fulfilled by having the component keep a checkpoint that reflects the state it was in when it last released a message, return value or output, an approach called checkpoint-on-send. However, there is a lot of flexibility. Taking full checkpoints can be time consuming, so the component may take incremental checkpoints. It may also amortize the cost of the checkpoint by waiting until it has accumulated several items to release.

Rule 1 can be satisfied without checkpoints in some special circumstances. A reliable component, perhaps one that is mirrored by one in a different failure domain can send messages at any time, as can a component with no application state of its own. In both cases, these components must still enforce Rules 2-4, reducing the benefit of avoiding the checkpoint.

Checkpoints and information on return values and processed and released messages specified in Rules 2-4 must be stored reliably. Losing this information requires out-of-band recovery that is likely to result in data inconsistencies.

Rule 2: ensure that every message it releases can reach the intended recipient at least once

A sending component has no way to guarantee that the receiving component processes a message. The best the sending component can do is to be sure that the message eventually gets delivered. A solution to this problem is for the sending component to keep releasing retries until the recipient sends a notification telling the sender to stop retrying the message. Since we assume network partitions eventually heal, one of the retries will eventually get to its destination. This solution requires that outgoing messages survive crashes of the sending component. That's not a problem if the component takes checkpoints. Just include outgoing messages that need to be retried in the checkpoint as shown in the Appendix.

Other approaches can be used. For example, the sender can deliver the message to reliable messaging infrastructure. Care is needed, though. Separating the application checkpoint from the persisting of messages risks failures that cause them to be inconsistent with each other. Hence, we do not recommend this approach.

Rule 3: put every received message in a state where it can be processed until it has been processed.

Rule 2 guarantees eventual, at least once delivery of every message. Rule 3 guarantees at most once processing of each message. The combination of these rules does not guarantee exactly once processing of each released message, but both existing implementations do [2, 3].

There are several ways to enforce Rule 3. One is to ensure that a received message can survive a crash and to discard subsequent retries until the message is processed. Another approach is to assign the same unique identifier to a message and its retries. The receiving process can keep track of which messages it has previously processed and deal with subsequent retries in accordance with Rule 4. Received messages don't need to survive crashes with this approach, because the guarantee can be met when a retry is received following restart.

An alternate approach is to transport the message on a reliable messaging bus, but there is a risk. The default behavior of many existing messaging buses is to discard the message once it has been delivered to the application. Should the receiving component crash before the message is processed, the effect would be the same as a permanently lost message. Hence, we do not recommend this approach.

Rule 4: return the same result on every retry

Return values are not retried. Receiving a retry of a message is the only way a component can know that the return value it released might not have been processed. Hence, it is important that the component that processed the message release the same return value every time it receives a retry of a message. This rule can be burdensome if a component must keep track of many return values, especially if they are large. A simple approach is to send the required data as new messages and release return values with minimal or no content. Rule 2 then guarantees that the message eventually reaches its destination. Alternatively, enforcing pairwise FIFO message processing allows discarding return values once a subsequent message has been received from that sender.

Rule 5: process each input no more than once.

Inputs may be lost without violating output validity. Whether inputs are lost in the network before reaching the in-COVR component's driver, because the in-COVR component is down when the message arrives, or because the in-COVR component crashed while processing the input does not affect output validity. However, processing an input more than once because of a crash could violate output validity. The simplest approach is for the in-COVR component's driver to keep inputs in a state where they will be lost should the in-COVR component crash.

Rule 6: make at least one attempt to release each output.

While an out-COVR component must try to release each output, it can crash while releasing an output or after the output has been released but before state denoting that fact can be recorded. In each case, the out-COVR component won't know if the output was released in its entirety or, in many cases, whether it was received and processed [4]. Whether at-most-once or at-least-once output is more appropriate depends on context known only to the driver. Hence, the driver must be allowed to determine when an output is considered to be complete.

5. Additional Considerations

An implementation can be fully compliant with COVR with just what has been described so far. However, there are additional considerations that can reduce the chance of incorrect implementations. Which ones are relevant depends on whether the COVR rules are enforced in the application, in its messaging libraries or the messaging infrastructure [2, 3].

We start with some additional definitions.

out-COVR-F: A special out-COVR component that is allowed to produce outputs containing information about COVR'd crashes

in-COVR-F: A special in-COVR component that receives inputs about COVR'd crashes to enable managing COVR components from within the COVR world

Reduce the chance that information about a COVR'd crash will appear in an output

Any outputs that contain information about COVR'd crashes violate output-validity. One way to prevent such outputs is to keep COVR components from knowing anything about COVR'd crashes [2]. However, this information is often critical for managing systems, so this option requires the use of external monitoring tools.

As an alternative, this guideline provides for an out-COVR-F component that is allowed to output information about COVR'd crashes. In order to avoid affecting application outputs, the output from an out-COVR-F component should be on a different channel than application output. For example, if out-COVR components send application output to stdout, the out-COVR-F component should put its output on stderr [3].

It is also advisable to reduce the chance that information about COVR'd crashes reaches an out-COVR component. Minimizing the number of out-COVR-F components, perhaps to a single one per management domain, may make it easier for application components to distinguish out-COVR components from out-COVR-F components. We can reduce the chance that application state will become dependent on information of COVR'd crashes by having out-COVR-F components only send stateless return values to COVR components. Output from out-COVR-F components may be used as input to in-COVR-F components but not in-COVR components. Finally, we recommend that other COVR components not share state with out-COVR-F and in-COVR-F components. .

Make in-COVR components highly available

It is common for inputs not to be replayed, which means that any such input received while the in-COVR component is inaccessible may be lost. Losing those inputs may lead to data inconsistencies between the COVR world and the outside world. Because a failure during processing an input may result in the input being lost, the in-COVR component should be reliable. Inputs might also be lost because the input queues of the in-COVR components fill up. We can mitigate both of these concerns by having the in-COVR component do as little work as possible before releasing the input in a message to a COVR component.

Simplify out-COVR components

Retrying outputs by the out-COVR component depends on whether the output should be at-least-once or at-most-once, but messages to other COVR components must be retried. An out-COVR component may be simpler if it sends return values but not messages to other COVR components. This guideline may be less important if COVR is implemented in the infrastructure.

Simplify in-COVR components

In an implementation where messages between COVR components are retried, the recipients must keep track of which messages have already been processed. In cases where inputs are not retried, there is no need to keep track of them. An in-COVR component can be simpler if it receives only return values from other COVR components. Following this guideline may be less important if COVR is implemented in the infrastructure.

Implement COVR in the infrastructure or a messaging library

As can be seen, there are a number of subtleties that need to be handled carefully when implementing COVR. That's why both waterken [2] and CKen [3] implement COVR as a part of the infrastructure. That may not always be possible, *i.e.*, when using an enterprise services bus for the messaging infrastructure. In this case what's needed is a send/receive library that separates preparing messages and releasing them and that considers a message to have been delivered only when processing of it is complete.

6. Conclusions

Distributed systems, even those made up of fully deterministic components, are almost never fully deterministic. Different runs against identical data can and do produce different outputs due to message arrival order differences, which raises the question of what constitutes the “right” answer. Output-validity says the “right” answer is any answer that could have been produced in a crash-free run. COVR guarantees this property.

Our discussion of COVR centers largely on crash/restart, but network issues introduce many of the same problems [4]. COVR addresses them both because Rule 1, which requires a component to restart from a crash in a state consistent with all messages, return values and outputs it has released, allows us to model network problems as crash/restarts. For example, a retry lost in the network can be modeled as a crash that occurred during processing of the retry, because neither results in a persistent state change or any new messages. A network delay can be modeled as a component that computes slowly or that crashes and recovers before completely processing the message. A network partition can be modeled as a message that causes the receiving component to crash every time it processes that message.

Today's distributed systems are likely to consist of independently written subsystems. There is often a substantial integration effort when assembling these subsystems into a composite solution. The composability property of COVR guarantees that accounting for crash-restart failures won't unduly complicate the assembly process as long as each subsystem is COVR-compliant and the components of the subsystems communicate with a COVR-compliant protocol.

COVR can be implemented in the infrastructure, a library that integrates checkpoints and messaging, or the application. We recommend the first of these based on our experience with waterken [2] and CKen [3]. Putting COVR in the infrastructure allows developers to design their applications as if crash-restart failures and temporary network partitions never occur.

Following just the six rules spelled out in this paper guarantees composable output-validity. The result is distributed systems that are easier to make resilient to crashes and temporary network partitions when building, maintaining, and assembling composite systems [5].

Acknowledgements: We'd like to thank the FRIAM group, particularly Daira Hopwood, Chip Morningstar, and Norm Hardy for their insightful comments.

References

1. http://en.wikipedia.org/wiki/Three-phase_commit_protocol
2. T. Close, <http://waterken.sourceforge.net/>
3. Sunghwan Yoo, Charles Killian, Terence Kelly, Hyoun Kyu Cho, and Steven Plite. "Composable Reliability for Asynchronous Systems: Treating Failures as Slow Processes", In proceedings of 2012 USENIX Annual Technical Conference (USENIX ATC '12). Boston, MA. 13-15 June, 2012, <http://ai.eecs.umich.edu/~tpkelly/Ken/>
4. P. Helland, "Idempotence is Not a Medical Condition", Comm. ACM, **9**, #5, pp. 56-65, May 2012.
5. M. Stiegler, "A Reliable and Secure Application Spanning Multiple Administrative Domains", HP Labs Technical Report HPL-2010-21, 2010

Appendix

The Figure in this Appendix sketches a COVR-compliant algorithm [5] that has been used in a production system [2]. There are two, concurrent operations. The first processes incoming messages. It collects messages until a checkpoint is complete, at which point the collected messages are added to the list of pending messages. The second operation works through the list of pending messages, releasing each until it receives a notification. You'll note from the first operation that a notification for the message is not sent until the effects of processing it are in a checkpoint. On restarting from a failure, a component restores its state and initializes the list of pending messages with the messages in its recovery state.

```
concurrently {
  concurrently for each component {
    for each received message {
      if already processed
      then return previous return value
    else {
      while processing message {
        if new message to be sent
        then add to list of outgoing messages
      }
      checkpoint app state, notifications, pending and outgoing messages
      delete old checkpoint
      notify sender
      append list of outgoing messages to list of pending messages
      empty list of outgoing messages
    }
  }
  concurrently for each component {
    for each pending message {
      if notification received
      then remove from list of pending messages
      else release pending message
    }
  }
}
on component restart {
  restore app state from checkpoint
  append outgoing messages to list of pending messages
  begin processing and messaging loops
}
```

A COVR algorithm.