

## Motivations

We rely on both Go and Kafka a lot at Segment. Unfortunately, the state of the Go client libraries for Kafka at the time of this writing was not ideal. The available options were:

- sarama, which is by far the most popular but is quite difficult to work with. It is poorly documented, the API exposes low level concepts of the Kafka protocol, and it doesn't support recent Go features like contexts. It also passes all values as pointers which causes large numbers of dynamic memory allocations, more frequent garbage collections, and higher memory usage.
- confluent-kafka-go is a cgo based wrapper around librdkafka, which means it introduces a dependency to a C library on all Go code that uses the package. It has much better documentation than sarama but still lacks support for Go contexts.
- goka is a more recent Kafka client for Go which focuses on a specific usage pattern. It provides abstractions for using Kafka as a message passing bus between services rather than an ordered log of events, but this is not the typical use case of Kafka for us at Segment. The package also depends on sarama for all interactions with Kafka.

This is where [kafka-go](#) comes into play. It provides both low and high level APIs for interacting with Kafka, mirroring concepts and implementing interfaces of the Go standard library to make it easy to use and integrate with existing software.

**Note:** In order to better align with our newly adopted Code of Conduct, the kafka-go project has renamed our default branch to [main](#). For the full details of our Code Of Conduct see this document.

## Kafka versions

[kafka-go](#) is currently tested with Kafka versions 0.10.1.0 to 2.7.1. While it should also be compatible with later versions, newer features available in the Kafka API may not yet be implemented in the client.

## Go versions

[kafka-go](#) requires Go version 1.15 or later.

---

## Connection

The `Conn` type is the core of the `kafka-go` package. It wraps around a raw network connection to expose a low-level API to a Kafka server.

Here are some examples showing typical use of a connection object:

```
1 // to produce messages
2 topic := "my-topic"
3 partition := 0
4
5 conn, err := kafka.DialLeader(context.Background(), "tcp", "localhost
6 :9092", topic, partition)
7 if err != nil {
8     log.Fatal("failed to dial leader:", err)
9 }
10
11 conn.SetWriteDeadline(time.Now().Add(10*time.Second))
12 _, err = conn.WriteMessages(
13     kafka.Message{Value: []byte("one!")},
14     kafka.Message{Value: []byte("two!")},
15     kafka.Message{Value: []byte("three!")},
16 )
17 if err != nil {
18     log.Fatal("failed to write messages:", err)
19 }
20
21 if err := conn.Close(); err != nil {
22     log.Fatal("failed to close writer:", err)
23 }
```

```
1 // to consume messages
2 topic := "my-topic"
3 partition := 0
4
5 conn, err := kafka.DialLeader(context.Background(), "tcp", "localhost
6 :9092", topic, partition)
7 if err != nil {
8     log.Fatal("failed to dial leader:", err)
9 }
10
11 conn.SetReadDeadline(time.Now().Add(10*time.Second))
12 batch := conn.ReadBatch(10e3, 1e6) // fetch 10KB min, 1MB max
13
14 b := make([]byte, 10e3) // 10KB max per message
15 for {
16     n, err := batch.Read(b)
17     if err != nil {
18         break
19     }
20 }
```

---

```
19     fmt.Println(string(b[:n]))
20 }
21
22 if err := batch.Close(); err != nil {
23     log.Fatal("failed to close batch:", err)
24 }
25
26 if err := conn.Close(); err != nil {
27     log.Fatal("failed to close connection:", err)
28 }
```

## To Create Topics

By default kafka has the `auto.create.topics.enable='true'` (KAFKA\_CFG\_AUTO\_CREATE\_TOPICS\_ENABLE='true' in the bitnami/kafka kafka docker image). If this value is set to 'true' then topics will be created as a side effect of `kafka.DialLeader` like so:

```
1 // to create topics when auto.create.topics.enable='true'
2 conn, err := kafka.DialLeader(context.Background(), "tcp", "localhost:9092", "my-topic", 0)
3 if err != nil {
4     panic(err.Error())
5 }
```

If `auto.create.topics.enable='false'` then you will need to create topics explicitly like so:

```
1 // to create topics when auto.create.topics.enable='false'
2 topic := "my-topic"
3
4 conn, err := kafka.Dial("tcp", "localhost:9092")
5 if err != nil {
6     panic(err.Error())
7 }
8 defer conn.Close()
9
10 controller, err := conn.Controller()
11 if err != nil {
12     panic(err.Error())
13 }
14 var controllerConn *kafka.Conn
15 controllerConn, err = kafka.Dial("tcp", net.JoinHostPort(controller.Host, strconv.Itoa(controller.Port)))
16 if err != nil {
17     panic(err.Error())
18 }
19 defer controllerConn.Close()
20
```

---

```
21
22 topicConfigs := []kafka.TopicConfig{
23     {
24         Topic:          topic,
25         NumPartitions:   1,
26         ReplicationFactor: 1,
27     },
28 }
29
30 err = controllerConn.CreateTopics(topicConfigs...)
31 if err != nil {
32     panic(err.Error())
33 }
```

### To Connect To Leader Via a Non-leader Connection

```
1 // to connect to the kafka leader via an existing non-leader connection
  rather than using DialLeader
2 conn, err := kafka.Dial("tcp", "localhost:9092")
3 if err != nil {
4     panic(err.Error())
5 }
6 defer conn.Close()
7 controller, err := conn.Controller()
8 if err != nil {
9     panic(err.Error())
10 }
11 var connLeader *kafka.Conn
12 connLeader, err = kafka.Dial("tcp", net.JoinHostPort(controller.Host,
    strconv.Itoa(controller.Port)))
13 if err != nil {
14     panic(err.Error())
15 }
16 defer connLeader.Close()
```

### To list topics

```
1 conn, err := kafka.Dial("tcp", "localhost:9092")
2 if err != nil {
3     panic(err.Error())
4 }
5 defer conn.Close()
6
7 partitions, err := conn.ReadPartitions()
8 if err != nil {
9     panic(err.Error())
10 }
```

---

```

11
12 m := map[string]struct{}{}
13
14 for _, p := range partitions {
15     m[p.Topic] = struct{}{}
16 }
17 for k := range m {
18     fmt.Println(k)
19 }

```

Because it is low level, the `Conn` type turns out to be a great building block for higher level abstractions, like the `Reader` for example.

## Reader

A `Reader` is another concept exposed by the `kafka-go` package, which intends to make it simpler to implement the typical use case of consuming from a single topic-partition pair. A `Reader` also automatically handles reconnections and offset management, and exposes an API that supports asynchronous cancellations and timeouts using Go contexts.

Note that it is important to call `Close()` on a `Reader` when a process exits. The kafka server needs a graceful disconnect to stop it from continuing to attempt to send messages to the connected clients. The given example will not call `Close()` if the process is terminated with `SIGINT` (ctrl-c at the shell) or `SIGTERM` (as docker stop or a kubernetes restart does). This can result in a delay when a new reader on the same topic connects (e.g. new process started or new container running). Use a `signal.Notify` handler to close the reader on process shutdown.

```

1 // make a new reader that consumes from topic-A, partition 0, at offset
  42
2 r := kafka.NewReader(kafka.ReaderConfig{
3     Brokers: []string{"localhost:9092", "localhost:9093", "localhost:9094"},
4     Topic:   "topic-A",
5     Partition: 0,
6     MaxBytes: 10e6, // 10MB
7 })
8 r.SetOffset(42)
9
10 for {
11     m, err := r.ReadMessage(context.Background())
12     if err != nil {
13         break
14     }
15     fmt.Printf("message at offset %d: %s = %s\n", m.Offset, string(m.Key), string(m.Value))
16 }

```

---

```
17
18 if err := r.Close(); err != nil {
19     log.Fatal("failed to close reader:", err)
20 }
```

## Consumer Groups

`kafka-go` also supports Kafka consumer groups including broker managed offsets. To enable consumer groups, simply specify the `GroupID` in the `ReaderConfig`.

`ReadMessage` automatically commits offsets when using consumer groups.

```
1 // make a new reader that consumes from topic-A
2 r := kafka.NewReader(kafka.ReaderConfig{
3     Brokers:  []string{"localhost:9092", "localhost:9093", "localhost:9094"},
4     GroupID:   "consumer-group-id",
5     Topic:     "topic-A",
6     MaxBytes:  10e6, // 10MB
7 })
8
9 for {
10     m, err := r.ReadMessage(context.Background())
11     if err != nil {
12         break
13     }
14     fmt.Printf("message at topic/partition/offset %v/%v/%v: %s = %s\n",
15         m.Topic, m.Partition, m.Offset, string(m.Key), string(m.Value))
16 }
17 if err := r.Close(); err != nil {
18     log.Fatal("failed to close reader:", err)
19 }
```

There are a number of limitations when using consumer groups:

- `(*Reader).SetOffset` will return an error when `GroupID` is set
- `(*Reader).Offset` will always return `-1` when `GroupID` is set
- `(*Reader).Lag` will always return `-1` when `GroupID` is set
- `(*Reader).ReadLag` will return an error when `GroupID` is set
- `(*Reader).Stats` will return a partition of `-1` when `GroupID` is set

## Explicit Commits

`kafka-go` also supports explicit commits. Instead of calling `ReadMessage`, call `FetchMessage` followed by `CommitMessages`.

---

```

1 ctx := context.Background()
2 for {
3     m, err := r.FetchMessage(ctx)
4     if err != nil {
5         break
6     }
7     fmt.Printf("message at topic/partition/offset %v/%v/%v: %s = %s\n",
8         m.Topic, m.Partition, m.Offset, string(m.Key), string(m.Value))
9     if err := r.CommitMessages(ctx, m); err != nil {
10         log.Fatal("failed to commit messages:", err)
11     }
12 }

```

When committing messages in consumer groups, the message with the highest offset for a given topic/partition determines the value of the committed offset for that partition. For example, if messages at offset 1, 2, and 3 of a single partition were retrieved by call to `FetchMessage`, calling `CommitMessages` with message offset 3 will also result in committing the messages at offsets 1 and 2 for that partition.

## Managing Commits

By default, `CommitMessages` will synchronously commit offsets to Kafka. For improved performance, you can instead periodically commit offsets to Kafka by setting `CommitInterval` on the `ReaderConfig`.

```

1 // make a new reader that consumes from topic-A
2 r := kafka.NewReader(kafka.ReaderConfig{
3     Brokers: []string{"localhost:9092", "localhost:9093", "localhost:9094"},
4     GroupID: "consumer-group-id",
5     Topic:   "topic-A",
6     MaxBytes: 10e6, // 10MB
7     CommitInterval: time.Second, // flushes commits to Kafka every
8     second
9 })

```

## Writer reference

To produce messages to Kafka, a program may use the low-level `Conn` API, but the package also provides a higher level `Writer` type which is more appropriate to use in most cases as it provides additional features:

- Automatic retries and reconnections on errors.

- 
- Configurable distribution of messages across available partitions.
  - Synchronous or asynchronous writes of messages to Kafka.
  - Asynchronous cancellation using contexts.
  - Flushing of pending messages on close to support graceful shutdowns.
  - Creation of a missing topic before publishing a message. *Note!* it was the default behaviour up to the version `v0.4.30`.

```
1 // make a writer that produces to topic-A, using the least-bytes
  distribution
2 w := &kafka.Writer{
3     Addr:      kafka.TCP("localhost:9092", "localhost:9093", "localhost
      :9094"),
4     Topic:     "topic-A",
5     Balancer: &kafka.LeastBytes{},
6 }
7
8 err := w.WriteMessages(context.Background(),
9     kafka.Message{
10         Key: []byte("Key-A"),
11         Value: []byte("Hello World!"),
12     },
13     kafka.Message{
14         Key: []byte("Key-B"),
15         Value: []byte("One!"),
16     },
17     kafka.Message{
18         Key: []byte("Key-C"),
19         Value: []byte("Two!"),
20     },
21 )
22 if err != nil {
23     log.Fatal("failed to write messages:", err)
24 }
25
26 if err := w.Close(); err != nil {
27     log.Fatal("failed to close writer:", err)
28 }
```

### Missing topic creation before publication

```
1 // Make a writer that publishes messages to topic-A.
2 // The topic will be created if it is missing.
3 w := &Writer{
4     Addr:      kafka.TCP("localhost:9092", "localhost:9093
      ", "localhost:9094"),
5     Topic:     "topic-A",
6     AllowAutoTopicCreation: true,
```



---

```

 7 }
 8
 9 messages := []kafka.Message{
10     {
11         Key: []byte("Key-A"),
12         Value: []byte("Hello World!"),
13     },
14     {
15         Key: []byte("Key-B"),
16         Value: []byte("One!"),
17     },
18     {
19         Key: []byte("Key-C"),
20         Value: []byte("Two!"),
21     },
22 }
23
24 var err error
25 const retries = 3
26 for i := 0; i < retries; i++ {
27     ctx, cancel := context.WithTimeout(context.Background(), 10*time.
28         Second)
29     defer cancel()
30     // attempt to create topic prior to publishing the message
31     err = w.WriteMessages(ctx, messages...)
32     if errors.Is(err, kafka.LeaderNotAvailable) || errors.Is(err,
33         context.DeadlineExceeded) {
34         time.Sleep(time.Millisecond * 250)
35         continue
36     }
37     if err != nil {
38         log.Fatalf("unexpected error %v", err)
39     }
40     break
41 }
42
43 if err := w.Close(); err != nil {
44     log.Fatal("failed to close writer:", err)
45 }
```

## Writing to multiple topics

Normally, the `WriterConfig.Topic` is used to initialize a single-topic writer. By excluding that particular configuration, you are given the ability to define the topic on a per-message basis by setting `Message.Topic`.

---

---

```

1 w := &kafka.Writer{
2     Addr:      kafka.TCP("localhost:9092", "localhost:9093", "localhost:9094"),
3     // NOTE: When Topic is not defined here, each Message must define it instead.
4     Balancer: &kafka.LeastBytes{},
5 }
6
7 err := w.WriteMessages(context.Background(),
8     // NOTE: Each Message has Topic defined, otherwise an error is returned.
9     kafka.Message{
10         Topic: "topic-A",
11         Key:    []byte("Key-A"),
12         Value: []byte("Hello World!"),
13     },
14     kafka.Message{
15         Topic: "topic-B",
16         Key:    []byte("Key-B"),
17         Value: []byte("One!"),
18     },
19     kafka.Message{
20         Topic: "topic-C",
21         Key:    []byte("Key-C"),
22         Value: []byte("Two!"),
23     },
24 )
25 if err != nil {
26     log.Fatal("failed to write messages:", err)
27 }
28
29 if err := w.Close(); err != nil {
30     log.Fatal("failed to close writer:", err)
31 }

```

**NOTE:** These 2 patterns are mutually exclusive, if you set `Writer.Topic`, you must not also explicitly define `Message.Topic` on the messages you are writing. The opposite applies when you do not define a topic for the writer. The `Writer` will return an error if it detects this ambiguity.

### Compatibility with other clients

**Sarama** If you're switching from Sarama and need/want to use the same algorithm for message partitioning, you can either use the `kafka.Hash` balancer or the `kafka.ReferenceHash` balancer: `* kafka.Hash = sarama.NewHashPartitioner` `* kafka.ReferenceHash = sarama.NewReferenceHashPartitioner`

The `kafka.Hash` and `kafka.ReferenceHash` balancers would route messages to the same par-

---

titions that the two aforementioned Sarama partitioners would route them to.

```
1 w := &kafka.Writer{
2     Addr:      kafka.TCP("localhost:9092", "localhost:9093", "localhost
      :9094"),
3     Topic:     "topic-A",
4     Balancer:  &kafka.Hash{},
5 }
```

**librdkafka and confluent-kafka-go** Use the `kafka.CRC32Balancer` balancer to get the same behaviour as librdkafka's default `consistent_random` partition strategy.

```
1 w := &kafka.Writer{
2     Addr:      kafka.TCP("localhost:9092", "localhost:9093", "localhost
      :9094"),
3     Topic:     "topic-A",
4     Balancer:  kafka.CRC32Balancer{},
5 }
```

**Java** Use the `kafka.Murmur2Balancer` balancer to get the same behaviour as the canonical Java client's default partitioner. Note: the Java class allows you to directly specify the partition which is not permitted.

```
1 w := &kafka.Writer{
2     Addr:      kafka.TCP("localhost:9092", "localhost:9093", "localhost
      :9094"),
3     Topic:     "topic-A",
4     Balancer:  kafka.Murmur2Balancer{},
5 }
```

## Compression

Compression can be enabled on the `Writer` by setting the `Compression` field:

```
1 w := &kafka.Writer{
2     Addr:      kafka.TCP("localhost:9092", "localhost:9093", "
      localhost:9094"),
3     Topic:     "topic-A",
4     Compression: kafka.Snappy,
5 }
```

The `Reader` will by determine if the consumed messages are compressed by examining the message attributes. However, the package(s) for all expected codecs must be imported so that they get loaded correctly.

---

*Note: in versions prior to 0.4 programs had to import compression packages to install codecs and support reading compressed messages from kafka. This is no longer the case and import of the compression packages are now no-ops.*

## TLS Support

For a bare bones Conn type or in the Reader/Writer configs you can specify a dialer option for TLS support. If the TLS field is nil, it will not connect with TLS. *Note:* Connecting to a Kafka cluster with TLS enabled without configuring TLS on the Conn/Reader/Writer can manifest in opaque `io.ErrUnexpectedEOF` errors.

## Connection

```
1 dialer := &kafka.Dialer{
2     Timeout: 10 * time.Second,
3     DualStack: true,
4     TLS:      &tls.Config{...tls config...},
5 }
6
7 conn, err := dialer.DialContext(ctx, "tcp", "localhost:9093")
```

## Reader

```
1 dialer := &kafka.Dialer{
2     Timeout: 10 * time.Second,
3     DualStack: true,
4     TLS:      &tls.Config{...tls config...},
5 }
6
7 r := kafka.NewReader(kafka.ReaderConfig{
8     Brokers: []string{"localhost:9092", "localhost:9093", "localhost:9094"},
9     GroupID: "consumer-group-id",
10    Topic:   "topic-A",
11    Dialer:  dialer,
12 })
```

## Writer

Direct Writer creation

---

```

1 w := kafka.Writer{
2     Addr: kafka.TCP("localhost:9092", "localhost:9093", "localhost:9094"),
3     Topic: "topic-A",
4     Balancer: &kafka.Hash{},
5     Transport: &kafka.Transport{
6         TLS: &tls.Config{},
7     },
8 }

```

Using `kafka.NewWriter`

```

1 dialer := &kafka.Dialer{
2     Timeout: 10 * time.Second,
3     DualStack: true,
4     TLS: &tls.Config{...tls config...},
5 }
6
7 w := kafka.NewWriter(kafka.WriterConfig{
8     Brokers: []string{"localhost:9092", "localhost:9093", "localhost:9094"},
9     Topic: "topic-A",
10    Balancer: &kafka.Hash{},
11    Dialer: dialer,
12 })

```

Note that `kafka.NewWriter` and `kafka.WriterConfig` are deprecated and will be removed in a future release.

## SASL Support

You can specify an option on the `Dialer` to use SASL authentication. The `Dialer` can be used directly to open a `Conn` or it can be passed to a `Reader` or `Writer` via their respective configs. If the `SASLMechanism` field is `nil`, it will not authenticate with SASL.

### SASL Authentication Types

#### Plain

```

1 mechanism := plain.Mechanism{
2     Username: "username",
3     Password: "password",
4 }

```

#### SCRAM

```

1 mechanism, err := scram.Mechanism(scram.SHA512, "username", "password")
2 if err != nil {

```

---

```
3     panic(err)
4 }
```

## Connection

```
1 mechanism, err := scram.Mechanism(scram.SHA512, "username", "password")
2 if err != nil {
3     panic(err)
4 }
5
6 dialer := &kafka.Dialer{
7     Timeout:      10 * time.Second,
8     DualStack:    true,
9     SASLMechanism: mechanism,
10 }
11
12 conn, err := dialer.DialContext(ctx, "tcp", "localhost:9093")
```

## Reader

```
1 mechanism, err := scram.Mechanism(scram.SHA512, "username", "password")
2 if err != nil {
3     panic(err)
4 }
5
6 dialer := &kafka.Dialer{
7     Timeout:      10 * time.Second,
8     DualStack:    true,
9     SASLMechanism: mechanism,
10 }
11
12 r := kafka.NewReader(kafka.ReaderConfig{
13     Brokers:      []string{"localhost:9092", "localhost:9093", "localhost:9094"},
14     GroupID:      "consumer-group-id",
15     Topic:        "topic-A",
16     Dialer:       dialer,
17 })
```

## Writer

```
1 mechanism, err := scram.Mechanism(scram.SHA512, "username", "password")
2 if err != nil {
3     panic(err)
4 }
```

---

```

5
6 // Transports are responsible for managing connection pools and other
  resources,
7 // it's generally best to create a few of these and share them across
  your
8 // application.
9 sharedTransport := &kafka.Transport{
10     SASL: mechanism,
11 }
12
13 w := kafka.Writer{
14     Addr:      kafka.TCP("localhost:9092", "localhost:9093", "localhost
        :9094"),
15     Topic:     "topic-A",
16     Balancer:   &kafka.Hash{},
17     Transport: sharedTransport,
18 }

```

## Client

```

1 mechanism, err := scram.Mechanism(scram.SHA512, "username", "password")
2 if err != nil {
3     panic(err)
4 }
5
6 // Transports are responsible for managing connection pools and other
  resources,
7 // it's generally best to create a few of these and share them across
  your
8 // application.
9 sharedTransport := &kafka.Transport{
10     SASL: mechanism,
11 }
12
13 client := &kafka.Client{
14     Addr:      kafka.TCP("localhost:9092", "localhost:9093", "localhost
        :9094"),
15     Timeout:   10 * time.Second,
16     Transport: sharedTransport,
17 }

```

### Reading all messages within a time range

```

1 startTime := time.Now().Add(-time.Hour)
2 endTime := time.Now()
3 batchSize := int(10e6) // 10MB
4
5 r := kafka.NewReader(kafka.ReaderConfig{

```

---

```

6   Brokers:  []string{"localhost:9092", "localhost:9093", "localhost
           :9094"},
7   Topic:    "my-topic1",
8   Partition: 0,
9   MaxBytes: batchSize,
10  })
11
12  r.SetOffsetAt(context.Background(), startTime)
13
14  for {
15      m, err := r.ReadMessage(context.Background())
16
17      if err != nil {
18          break
19      }
20      if m.Time.After(endTime) {
21          break
22      }
23      // TODO: process message
24      fmt.Printf("message at offset %d: %s = %s\n", m.Offset, string(m.
           Key), string(m.Value))
25  }
26
27  if err := r.Close(); err != nil {
28      log.Fatal("failed to close reader:", err)
29  }
```

## Logging

For visibility into the operations of the Reader/Writer types, configure a logger on creation.

### Reader

```

1  func logf(msg string, a ...interface{}) {
2      fmt.Printf(msg, a...)
3      fmt.Println()
4  }
5
6  r := kafka.NewReader(kafka.ReaderConfig{
7      Brokers:  []string{"localhost:9092", "localhost:9093", "
           localhost:9094"},
8      Topic:    "my-topic1",
9      Partition: 0,
10     Logger:    kafka.LoggerFunc(logf),
11     ErrorLogger: kafka.LoggerFunc(logf),
12 })
```



---

## Writer

```
1 func logf(msg string, a ...interface{}) {
2     fmt.Printf(msg, a...)
3     fmt.Println()
4 }
5
6 w := &kafka.Writer{
7     Addr:      kafka.TCP("localhost:9092"),
8     Topic:      "topic",
9     Logger:      kafka.LoggerFunc(logf),
10    ErrorLogger: kafka.LoggerFunc(logf),
11 }
```

## Testing

Subtle behavior changes in later Kafka versions have caused some historical tests to break, if you are running against Kafka 2.3.1 or later, exporting the `KAFKA_SKIP_NETTEST=1` environment variables will skip those tests.

Run Kafka locally in docker

```
1 docker-compose up -d
```

Run tests

```
1 KAFKA_VERSION=2.3.1 \
2 KAFKA_SKIP_NETTEST=1 \
3 go test -race ./...
```

(or) to clean up the cached test results and run tests:

```
1 go clean -cache && make test
```