KIP-388: Add observer interface to record request and response

Status

Current state: *Under Discussion*

Discussion thread: *here*

JIRA: *KAFKA-7596*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

A Kafka audit system is very useful for both users and developers of the streaming platform since it reveals system insights. For example, one question that could be answered with audit information is that which application consumes or produces how much data to which topic(s) over what period of time. Such Insights derived from audit information opens up opportunities. For example,

1. Traffic analysis. It allows detailed analysis of traffic on particular topics, for particular principles, or for specific request types. Useful information could be derived from the audit information. For example, user can easily figure out what topics have not been accessed for some certain period of time. With this information, users can initiate some topic deletion process and etc.
2. Correctness analysis and reporting. There could be various definitions on “correctness”. For example, “correctness” could be defined as whether the number of records produced to a Kafka cluster equals the number of records that is mirrored to another cluster.
3. SLO monitoring. The response time to a request could be broken down by topic, principle or any other arbitrary criteria.
4. Cost attribution. For example, the number of bytes produced or consumed could be broken down by topic, principle or any other arbitrary criteria.

Sources of an audit system are where the auditing information is initially generated. Kafka server could be a source since it can access every request and its response.

Public interfaces

Define a ObservableStruct interface:

ObservableStruct Interface

```
public interface ObservableStruct {

  public Byte get(Field.Int8 field);

  public Integer get(Field.Int32 field);

  public Long get(Field.Int64 field);

  public Short get(Field.Int16 field);

  public String get(Field.Str field);

  public String get(Field.NullableStr field);

  public Long getOrElse(Field.Int64 field, long alternative);

  public Short getOrElse(Field.Int16 field, short alternative);

  public Byte getOrElse(Field.Int8 field, byte alternative);

  public Integer getOrElse(Field.Int32 field, int alternative);

  public String getOrElse(Field.NullableStr field, String alternative);

  public String getOrElse(Field.Str field, String alternative);

  public boolean hasField(String name);

  public boolean hasField(Field def);

  public Byte getByte(BoundField field);

  public byte getByte(String name);

  public BaseRecords getRecords(String name);

  public Short getShort(BoundField field);

  public Short getShort(String name);

  public Integer getInt(BoundField field);

  public Integer getInt(String name);

  public Long getUnsignedInt(String name);

  public Long getLong(BoundField field);

  public Long getLong(String name);

  public String getString(BoundField field);

  public String getString(String name);

  public Boolean getBoolean(BoundField field);

  public Boolean getBoolean(String name);

  public ByteBuffer getBytes(BoundField field);

  public ByteBuffer getBytes(String name);

  public int sizeOf();
}
```

This interface defines the methods that can be used in the client's implementation of the observer to extract information from the request.

Change the Struct class to implement the ObservableStruct interface. Only "implements ObservableStruct" is added and no other change is required.

Struct Class

```
public class Struct implements ObservableStruct {
 // ...
}
```

Define ObservableStructImpl class which also implements the ObservableStruct interface:

ObservableStructImpl Class

```
public class ObservableStructImpl implements ObservableStruct {

  private final Struct struct;

  public ObservableStructImpl(Struct struct) {
    if (struct == null) {
      throw new IllegalStateException("Can not use null Strcut instance to construct ObservableStructImpl
instance.");
    }
    this.struct = struct;
  }

  public Byte get(Field.Int8 field) {
    return struct.get(field);
  }

  public Integer get(Field.Int32 field) {
    return struct.get(field);
  }

  public Long get(Field.Int64 field) {
    return struct.get(field);
  }

  public Short get(Field.Int16 field) {
    return struct.get(field);
  }

  public String get(Field.Str field) {
    return struct.get(field);
  }

  public String get(Field.NullableStr field) {
    return struct.get(field);
  }

  public Long getOrElse(Field.Int64 field, long alternative) {
    return struct.getOrElse(field, alternative);
  }

  public Short getOrElse(Field.Int16 field, short alternative) {
    return struct.getOrElse(field, alternative);
  }

  public Byte getOrElse(Field.Int8 field, byte alternative) {
    return struct.getOrElse(field, alternative);
  }

  public Integer getOrElse(Field.Int32 field, int alternative) {
    return struct.getOrElse(field, alternative);
  }

  public String getOrElse(Field.NullableStr field, String alternative) {
    return struct.getOrElse(field, alternative);
  }

  public String getOrElse(Field.Str field, String alternative) {
    return struct.getOrElse(field, alternative);
  }


  public boolean hasField(String name) {
    return struct.hasField(name);
  }

  public boolean hasField(Field def) {
    return struct.hasField(def);
  }

  public ObservableStructImpl getObservableStructImp(BoundField field) {
    return new ObservableStructImpl(struct.getStruct(field));
  }

  public ObservableStructImpl getObservableStructImp(String name) {
    return new ObservableStructImpl(struct.getStruct(name));
  }

  public Byte getByte(BoundField field) {
    return struct.getByte(field);
  }

  public byte getByte(String name) {
    return struct.getByte(name);
  }

  public BaseRecords getRecords(String name) {
    return struct.getRecords(name);
  }

  public Short getShort(BoundField field) {
    return struct.getShort(field);
  }

  public Short getShort(String name) {
    return struct.getShort(name);
  }

  public Integer getInt(BoundField field) {
    return struct.getInt(field);
  }

  public Integer getInt(String name) {
    return struct.getInt(name);
  }

  public Long getUnsignedInt(String name) {
    return struct.getUnsignedInt(name);
  }

  public Long getLong(BoundField field) {
    return struct.getLong(field);
  }

  public Long getLong(String name) {
    return struct.getLong(name);
  }

  public String getString(BoundField field) {
    return struct.getString(field);
  }

  public String getString(String name) {
    return struct.getString(name);
  }
  public Boolean getBoolean(BoundField field) {
    return struct.getBoolean(field);
  }

  public Boolean getBoolean(String name) {
    return struct.getBoolean(name);
  }

  public ByteBuffer getBytes(BoundField field) {
    return struct.getBytes(field);
  }

  public ByteBuffer getBytes(String name) {
    return struct.getBytes(name);
  }

  public int sizeOf() {
    return struct.sizeOf();
  }

  public ObservableStructImpl [] getObservableStructImplArray(BoundField field) {
    Object [] structObjs = struct.getArray(field);
    return convertStructArrayToObservableStructImplArray(structObjs);
  }

  public ObservableStructImpl [] getObservableStructImplArray(String name) {
    Object [] structObjs = struct.getArray(name);
    return convertStructArrayToObservableStructImplArray(structObjs);
  }

  private ObservableStructImpl [] convertStructArrayToObservableStructImplArray(Object [] structObjs) {
    ObservableStructImpl [] observableStructs = new ObservableStructImpl[structObjs.length];
    for (int i = 0; i < observableStructs.length; i++) {
      observableStructs[i] = new ObservableStructImpl((Struct)structObjs[i]);
    }
    return observableStructs;
  }
}
```

The instantiated instance of this class is used by the observer.

The class hierarchy of the Struct class is that the ObservableStruct interfaces define a set of getters and the interface is implemented by the Struct class and the ObservableStructImp class.

Define an Observer interface:

Observer Interface

```
 /** The Observer interface. This class allows user to implement their own auditing solution.
  * Notice that the Observer may be used by multiple threads, so the implementation should be thread safe. */
public interface Observer extends Configurable {

  /**
   * Record the request
   *
   * @param struct         Struct information in a request.
   * @param clientAddress  Client host IP information.
   * @param principal      Client principle from its request.
   * @param apiKey         Request type information.
   * @param apiVersion     Version of the request.
   * @param correlationId  An ID used to correspond a recordRequest method call to a recordResponse method call.
   */
  public void recordRequest(ObservableStructImpl struct,
                            InetAddress clientAddress,
                            KafkaPrincipal principal,
                            ApiKeys apiKey,
                            short apiVersion,
                            String correlationId);

  /**
   * Record the request
   *
   * @param struct Struct information in a response.
   * @param apiVersion  Version of the response.
   * @param correlationId  An ID used to correspond a recordRequest method call to a recordResponse method call.
   */
  public void recordResponse(ObservableStructImpl struct, short apiVersion, String correlationId);

  /**
   * Close the observer with timeout.
   *
   * @param timeout The maximum time to wait to close the observer.
   * @param unit    The time unit.
   */
  public void close(long timeout, TimeUnit unit);
}
```

Observers are instantiated by calling the no-arg constructor, then the configure() method is called, and then close() is called with no further recordRequest() nor recordResponse() calls expected.
With this interface and its given arguments, the information we can extract from every produce or fetch request is:

1. Topic name
2. Client principal
3. Client IP address
4. Number of bytes produced or consumed
5. Records count (if the batch version is 2 or greater, otherwise a deep decompression is required to get the count)

Proposed Changes

Add interfaces and implementations for classes that are described above and a configuration property in the KafkaConfig class to allow users to specify implementations of the observer interface. The configuration property lets user define a list of observer implementations which are going to be invoked.

Configuration property

```
/** ********* Broker-side Observer Configuration ****************/
val ObserverClassProp = "observer.class.names"
```

Add code to the broker (in SocketServer, RequestContext and AbstractResponse) to allow Kafka servers to invoke all observers defined on each request and each response. More specifically, initialize a list of observer instances in the SocketServer and pass them to RequestContext and AbstractResponse.
For example, in SocketServer:

Configuration property

```
// Initialize a list of observer instances using configuration and classes provided by user
private val observers =  config.getConfiguredInstances(KafkaConfig.ObserverClassesProp, classOf[Observer])
```

In RequestContext:

How request is recorded

```
// This method is called on every RequestContext instance and RequestContext instance is constructed when
SocketServer receives a request
public RequestAndSize parseRequest(ByteBuffer buffer) {
    if (isUnsupportedApiVersionsRequest()) {
        // Unsupported ApiVersion requests are treated as v0 requests and are not parsed
        ApiVersionsRequest apiVersionsRequest = new ApiVersionsRequest((short) 0, header.apiVersion());
        return new RequestAndSize(apiVersionsRequest, 0);
    } else {
        ApiKeys apiKey = header.apiKey();
        try {
            short apiVersion = header.apiVersion();
            Struct struct = apiKey.parseRequest(apiVersion, buffer);
            // Execute the observing logic only when there is at least on observer implementation is provided
            if (observers != null && observers.size() > 0) {
                // Convert Struct to ObservableStructImpl which is recorded by each observer
                ObservableStructImpl observableStruct = new ObservableStructImpl(struct);
                for (Observer observer : observers) {
                    observer.recordRequest(observableStruct, clientAddress, principal, apiKey, apiVersion,
                        connectionId);
                }
            }

            AbstractRequest body = AbstractRequest.parseRequest(apiKey, apiVersion, struct);
            return new RequestAndSize(body, struct.sizeOf());
        } catch (Throwable ex) {
            throw new InvalidRequestException("Error getting request for apiKey: " + apiKey +
                    ", apiVersion: " + header.apiVersion() +
                    ", connectionId: " + connectionId +
                    ", listenerName: " + listenerName +
                    ", principal: " + principal, ex);
        }
    }
}
```

In AbstractResponse:

How response is recorded

```
// This method is called on every response
protected Send toSend(String destination, ResponseHeader header, short apiVersion, List<Observer> observers) {
    return new NetworkSend(destination, serialize(destination, apiVersion, header, observers));
}

public ByteBuffer serialize(String connectionId, short version, ResponseHeader responseHeader, List<Observer>
observers) {
    Struct responseStruct = toStruct(version);

    // Execute the observing logic only when there is at least on observer implementation is provided
    if (observers != null && observers.size() > 0) {
        ObservableStructImpl observableStruct = new ObservableStructImpl(responseStruct);
        for (Observer observer : observers) {
            observer.recordResponse(observableStruct, version, connectionId);
        }
    }
    return serialize(responseHeader.toStruct(), responseStruct);
}
```

There are mostly two reasons why observers are invoked in these two classes:

1. Each request/response instance is passed through those functions so that the observer can truly intercepts each request/response.
2. It is in these functions (serialize() and parseRequest()) that the transformation from/to Struct happens. In other words, the natural lifecycle of Struct instance exists only in these functions. Therefore convert Struct into ObservableStructImpl which get passed to observers could only happen in these functions.

An example of how user could implement the recordRequest method and recordResponse method from the observer interface to record on the Produce request and Fetch response:

Observer Partial Implementation Example

```
public void recordRequest(ObservableStructImpl struct,
                          InetAddress clientAddress,
                          KafkaPrincipal principal,
                          ApiKeys apiKey,
                          short apiVersion,
                          String correlationId) {

  if (apiKey == ApiKeys.PRODUCE) {
    // The information extracted from the ObservableStructImpl
    Map<String, Integer> topicProduceBytesCount = new HashMap<>();

    for (ObservableStructImpl topicDataStruct : struct.getObservableStructImplArray("topic_data")) {

      String topic = topicDataStruct.get(TOPIC_NAME);

      for (ObservableStructImpl partitionResponseStruct : topicDataStruct.getObservableStructImplArray("data"))
{

        MemoryRecords records = (MemoryRecords) partitionResponseStruct.getRecords("record_set");

        if (topicProduceBytesCount.containsKey(topic)) {
          topicProduceBytesCount.put(topic, topicProduceBytesCount.get(topic) + records.sizeInBytes());
        } else {
          topicProduceBytesCount.put(topic, topicProduceBytesCount.get(topic) + records.sizeInBytes());
        }
      }
    }
  } else if (apiKey == ApiKeys.FETCH) {
        // ...
  }
}

public void recordResponse(ObservableStructImpl struct, short apiVersion, String correlationId) {

  // ...
  // Use correlationId to figure out to what request type this response is
  // ...
  // The information extracted from the ObservableStructImpl
  Map<String, Integer> topicFetchBytesCount = new HashMap<>();

  for (ObservableStructImpl topicDataStruct : struct.getObservableStructImplArray("responses")) {
    String topic = topicDataStruct.get(TOPIC_NAME);

    for (ObservableStructImpl partitionResponseStruct : topicDataStruct.getObservableStructImplArray
("partition_responses")) {
      BaseRecords baseRecords = partitionResponseStruct.getRecords("record_set");
      if (!(baseRecords instanceof MemoryRecords))
        throw new IllegalStateException("Unknown records type found: " + baseRecords.getClass());
      MemoryRecords records = (MemoryRecords) baseRecords;

      if (topicFetchBytesCount.containsKey(topic)) {
        topicFetchBytesCount.put(topic, topicFetchBytesCount.get(topic) + records.sizeInBytes());
      } else {
        topicFetchBytesCount.put(topic, topicFetchBytesCount.get(topic) + records.sizeInBytes());
      }
    }
  }
}
```

With ObservableStructImp, user of observer can interact with Struct in a read-only way. In the above example, the way information is extracted from the ObservableStructImpl is very similar to the "toStruct()" method for each specific request and response.

Rejected Alternatives for Interfaces

There are two set of rejected interfaces.

The first approach is to introduce one interface (with implementation) for each kind of AbstractRequest and AbstractResponse. Each interface would have a set of specific getters that are necessary to retrieve commonly important information from that request/response. For example, for a ProduceRequest, a getter to get a mapping from topic/partition to the number of bytes/records produced is commonly important. Then user can use those interfaces to retrieve request/response-specific information in their observer implementation. The issue with this approach is that the number of interfaces (with implementation) is too large due to the fact that we have about 40 requests and 40 responses. It is not trivial to define exactly what set of getters to be defined on each interface. Maintainability and extensibility are not ideal either. On the other side, the proposed approaches do not introduce 80 interfaces plus implementation and do not have much need to be extended potentially.

The second approach is to introduce a generic observer interface:

Type-specific Observer

```
public interface Observer <T extends AbstractRequest, R extends AbstractResponse> {
        // ...
        public void record(T request, R response);
        // ...
}
```

So that user can provide implementations such as:

Type-specific Observer Implementation

```
class ProduceObserver <ProduceRequest, ProduceResponse> {
        // ...
        public void record(ProduceRequest request, ProduceResponse response);
        // ...
}
```

In this approach, Observer implementation is type-specific which means there is one implementation of the observer interface corresponds to only one type of request/response. This approach is straight-forward and the interface is simple and clean. However the fatal issue is that users can cast the specific request/response implementation class to the abstract class which is considered as the internal classes that should not be exposed and freeze these into a public API would hinder future evolution of the project. The current proposed interfaces base on the Struct class which essentially represents the Kafka protocol format("wire format") which is public. With knowledge about the format of each request/response, user could extract information in a flexible way. Thus the proposed interfaces are unlikely to hinder the evolution of the project.

Compatibility, Deprecation, and Migration Plan

N/A. No observers are defined by default. Test of any other non-trivial implementation should be done by its implementers. Exceptions thrown by any of the observer(s) should be caught and the functionalities of the broker should not be affected.

Rejected Alternatives for Kafka Audit

Besides the broker-side auditing approach, the alternative approach is client-side auditing which means the auditor is embedded in the producer and consumer. From what we have learned in practice, the major issue of the client-side auditing approach which badly affects the usability of the auditing system is the lack of central administration on user’s selection of Kafka clients and versions of Kafka clients to use in their applications. Because there are many Kafka clients (even ones implemented in different programming languages such as Golang, C++, Python and etc) and versions. There are three direct consequences.

a. Lack of coverage due to the sheer number of users, slow deployment of diverse clients, and hard to enforce compliance.
b. In order to support diverse clients, various implementations of the auditor have to be provided which causes engineering overhead.
c. The slow rate of development on the audit system again due to the number of clients.

These existing Kafka server metrics provide lower-level insights into the system such as byte-in/out rates. However, an audit system complements the operation-level server metrics by providing additional application-level information such as byte-in/out rate associated with some specific application(s).

Since the major motivation is to support the auditing system, the interface could be named as “Auditor”. However, “Observer” is more general and flexible than it.

Another rejected alternative is to make an interceptor interface instead of an observer interface. Firstly, “inceptor” implies the possibility of modifying data which is not implied by an “observer”. Secondly, except the difference between interceptor and observer, the concept of the interceptor is more general than an observer in terms of what things could be done. However, the extra generalization complicates the KIP and is not necessary.