Schema based topics

The idea here is for topics to have associated a plug-able set of schema's that the Kafka broker will validate against when produced. The broker will also execute the plugged in logic based on that topic's associated field for plug-ins. This should be >= 1 so we can implement a pass through of iterations on the data prior to save (e.g. Security Authorizations).

To best facilitate this we could store <TBD> a list of schemaIdHash + schema. Every topic would have a class file associated with it to run the management of the schemes for the topic. The plugin could hold the schemas compiled or in another repository. The storage of the schemas should be key/value based [schemaIdHash] = schema. We may want to order and priortize these so that certain plug-in can iterator on the message before others (e. g. you should do authorizations first).

1) We are going to need a cli tool for the crud https://issues.apache.org/jira/browse/KAFKA-1694 and other things that exist today.

2) Besides validation of the schema on the producer side and keeping client compatibility with that we also need a way for consumers once subscribed to a topic (from a group perspective) to read in the key/value schema information. This could just be part of the OffsetResponse A Guide To The Kafka Protocol#OffsetResponse

3) We should lump the client compatibility kit (some thoughts on that https://github.com/stealthly/kafka-clients/wiki/Compatibility) work into this effort too.

5) This design also can work to implement Authorizations for the data in regards ACL (at least the security bits on the data to validate).

6) I think built in initial support for Avro would be awesome and probably account for the largest percentage of existing Kafka installations. We could use Camus encoders/decoders for avro https://github.com/linkedin/camus/blob/master/camus-kafka-coders/src/main/java/com/linkedin/camus/etl/kafka/coders/KafkaAvroMessageEncoder.java / https://github.com/linkedin/camus/blob/master/camus-kafka-coders/src/main/java/com/linkedin/camus/etl/kafka/coders/KafkaAvroMessageDecoder.java and json https://github.com/linkedin/camus/blob/master/camus-kafka-coders/src/main/java/com/linkedin/camus/etl/kafka/coders/JsonStringMessageDecoder.java but with a layer of faster xml databind over it e.g.

```
import com.fasterxml.jackson.databind.{DeserializationFeature, ObjectMapper}
import com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper
import com.fasterxml.jackson.module.scala.DefaultScalaModule
object JsonUtil {
  val mapper = new ObjectMapper() with ScalaObjectMapper
  mapper.registerModule(DefaultScalaModule)
  mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
  def toJson(value: Map[Symbol, Any]): String = {
    toJson(value map { case (k,v) => k.name -> v})
  }
  def toJson(value: Any): String = {
    mapper.writeValueAsString(value)
  }
  def toMap[V](json:String)(implicit m: Manifest[V]) = fromJson[Map[String,V]](json)
  def toObj[V](json:String)(implicit m: Manifest[V]) = fromJson[V](json)
  def toSeq[V](json:String)(implicit m: Manifest[V]) = fromJson[Seq[V]](json)
  def fromJson[T](json: String)(implicit m : Manifest[T]): T = {
    mapper.readValue[T](json)
  }
}
object MarshallableImplicits {
  implicit class Unmarshallable(unMarshallMe: String) {
    def toMapFrom: Map[String,Any] = JsonUtil.toMap(unMarshallMe)
    def toMapOf[V]()(implicit m: Manifest[V]): Map[String,V] = JsonUtil.toMap[V](unMarshallMe)
    def toObj[V]()(implicit m: Manifest[V]): V = JsonUtil.toObj[V](unMarshallMe)
    def toSeq[V]()(implicit m: Manifest[V]): Seq[V] = JsonUtil.toSeq[V](unMarshallMe)
    def fromJson[T]()(implicit m: Manifest[T]): T =  JsonUtil.fromJson[T](unMarshallMe)
  }
  implicit class Marshallable[T](marshallMe: T) {
    def toJson: String = JsonUtil.toJson(marshallMe)
  }
}
```

Then we can make our objects from json easily. Starting with this first is another approach it would definitly get things work better faster and we can refactor in avro to make sure it does support plugins.