Transports for knative eventing

What problems are we trying to solve?

Event importers, channels and other components need to speak multiple protocols.
Currently that is done by repetitive code in each importer or channel implementation.
Transport code is not well encapsulated from could that could be transport-independent.
Here are some use cases that could be improved:

1. An expert in a foreign protocol wants to build an event importer. They know how to convert between foreign events and cloudevents, but they don't know anything about knative CRDs, reconciliation, controllers etc.
2. Once someone has written a transport for protocol X, it should be trivial to build an importer, channel or any other component speaking protocol X.
3. Common code should be in libraries that can be maintained and updated, not in hand-modified generated code.
4. Transport code and knative orchestration code should be maintained and updated separately by the appropriate experts.

Go and non-Go components

For golang components we can standardize on the CNCF cloudevents Transport interface.
[1]This is a generic interface for sending and receiving events and already has several implementations (HTTP, NATS and AMQP)

For non-golang components we can still solve part of the problem by having the foreign expert implement *adapter executables* with a standard configuration interface based on JSON objects.

Adapter executables and transport configuration

An *adapter executable* can be written in any language (we will provide extra support for Go)
The executable reads JSON objects to configure receiver and/or sender transports from well-known environment variables: ADAPTER_SENDER and ADAPTER_RECEIVER. We may add other environment variables to control logging, enabling debug output etc.
Controllers that use adapter executables will construct JSON configuration from their CRD YAML spec and from the live status of running objects.

- Service entries in the configuration are resolved to URLs in the JSON
- Secrets in the configuration are resolved to mounts and file names in the JSON
- Transport-specific configuration is copied verbatim

---

[1]We should probably avoid command-line arguments to avoid the vagaries of different languages argument and option handling, and possible shell quoting issues.

Comparison to ContainerSource

ContainerSource takes a similar approach but it has explicit CRD entries for arguments and environment variables to pass to the adapter. The problems with this include:

- Can't be validated - there are no generic validation rules that can be applied to the env/arg section of the CRD
- Configuring *secrets* and passing them to the adapter is not straightforward. Tracked in #1151.

Golang adapter support

The *kntransport* library provides all the transport-independent code for an adapter. The foreign protocol expert only needs to:

- Define Go factory struct types to define transport-specific configuration.
- Define a New() method to create a transport from the configuration.
- Call a single function in main()

Examples:

- HTTP adapter using the existing cloudevents HTTP transport
- File adapter shows an artificial (but short) example of implementing a new transport. The example transport reads/writes JSON-events from files or stdin/out.

Controllers and CRDs (*Work in progress)*

The goal is to provide a reference CRD for importers, channels and other components, along with reference controller implementations that provide:

- Robust reconciliation.
- Monitoring of started containers.
- Handling errors, timeouts and retries.
- High-quality tracing, status and error logging for monitoring and debugging.

All *transport* work is delegated to an adapter executable.

Replace the source configuration

The reference spec includes:

- Source: HTTP listening address, *secrets* for inbound TLS. This section of the configuration will be *replaced* by new sources with appropriate listening/connecting/subscribing configuration for the source.
- Sink: (service, addressable, callable)

```
# Original reference CRD with HTTP listening source.
spec:
  source:
    servicePorts:
      - name: http
        port: 80
        targetPort: 8080
      - name: https
        port: 443
        targetPort: 8443
    secret:
      name: my-https-certificate
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1alpha1
      kind: Broker
      name: default
```

To create a new source: copy the reference spec and replace the source configuration. For example a hypothetical AMQP source:

```
# Modified CRD for AMQP source: the source configuration section
# has been replaced with AMQP connection config.
spec:
  source:
    connect:
      host: foobar.com
      port: 5672
      cert:
        secret:
          name: my-tls-certificate
    sourceLinks: ["interest1", "interest2"]
    # A full AMQP source would also allow listening. For more complete
    # options see connection and bridging

  # NOTE: sink section remains unchanged.
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1alpha1
      kind: Broker
      name: default
```

In a running source, the status includes a URI reconciled from sink:

```
status:
  sinkUri: https://my-event-source.default.svc.cluster.local/
```

The reference *controller* reconciles and pre-processes spec and status into a single JSON configuration for the *adapter image* so that the adapter image need not be aware of this process.
In the generated JSON config:

- *sink* is replaced by the reconciled sinkURI string from status.
- *secrets* are replaced by the string name of a mounted secret files.
- other spec configuration is included verbatim

For example from the AMQP CRD above we would get:

```
{
   "source": {
      "connect": {
         "host": "foobar.com",
         "port": 5672,
         "cert": "/mounted/secrets/place/my-tls-certificate.crt"
      },
      "sourceLinks": ["interest1", "interest2"]
   },
   "sink": {
      "sinkUri": "https://my-event-source.default.svc.cluster.local/"
   }
}
```

Secrets and security

Env vars and process args are insecure places to store sensitive information. The initial proposal here is to mount all secrets as volumes and pass file names to the adapter process. OS-level file protection will prevent unauthorized processes from reading the files.

Customizing the controller

TODO: reference controller decomposed into libraries that allow customized replacement of selected behaviors while using reference implementation for others.

Operators

TODO: operator for sources that use the reference source.

- Can we provide a sane “default operator”
- What kind of customization is needed.
- Do we want separate operators per source, or a central operator or both?

QoS and Acknowledgements

Different domains provide different QoS and acknowledgement schemes (0=unreliable, 1=at least once, 2=exactly once) the reference adapter needs to know what QoS it is dealing with.

Other features

- Live update to transport configuration (e.g. change topics in kafka)?
- Health and readiness probes
- Validation: webhook / OpenAPI

Appendix: Notes on existing sources

Most source controllers do the following:

1. check for an existing receiver adapter matching source config
2. if not found create a deployment for a new receiver adapter image.
3. Create event-types - only if the sink is a Broker.

Special cases:

- camel: creates "IntegrationPlatform" k8 object, not deployment.
- githubsource: creates a webhook (otherwise is like most controllers)
- kuberneteseventsource:
    - has extra checks before deploying adapter.
    - uses containersource to deploy adapter.

All sources have these conditions:

- Sink, Deployed, EventTypes (and corresponding "NoSink" etc.)

The following have extra conditions:

- camel, container: Deploying
- gcpubsub: Subscribed
- cronjobsource: InvalidSchedule, Schedule

For SourceKit I propose these “universal” conditions:

- Sink, Deployed, EventTypes, Ready

Ready and *NoReady* corresponds to “Subscribed” or “Schedule” events - source is ready to start *receiving* (or generating) events. Distinct from *Sink* - source is ready to start *sending*.

Some sources need secrets to authenticate/authorize with the domain transport, the way these are configured is not consistent.

Common config for source

```
spec:
    serviceAccountName:
      type: string
      description: "name of the ServiceAccount to use to run the receive adapter."
    sink:
      type: object
      description: "A reference to the object that should receive events."
  secretSomething: type object - not consistently named
  extras: github eventTypes, apiserversource API version to watch.
```

status:
sinkUri:
conditions: complex and repeated - why not encapsulated as type?
extras: github webHookIDKey:

StatefulSet (kafka) vs. Deployment