Improve Logging in APIs

Proposal: 0011
Authors: Dr. Brandon Wiley, Operator Foundation
Status: Implementation in process
Implementations

- Go - https://github.com/OperatorFoundation/shapeshifter-transports/tree/v3

Introduction

The PT 2.1 specification sections on language-specific APIs for Go and Swift (and a proposed one for Java) do not specify any mechanism for transports to log useful information for debugging to the dispatcher system log. There is an ad-hoc mechanism in place in the reference implementation of the dispatcher, but it is not standardized in the specification. This proposal introduces changes to the Go, Swift, and Java APIs to allow transports to log in a standardized way.

Motivation

Better logging is a requested feature from developers of both applications and transports. The API used to connect applications and transports offers only very basic error handling. An application can see if a transport has failed to initialize, but there is no mechanism to see why it failed. Transport developers are free to do their own transport-specific logging inside the dispatcher state directory. However, when testing a transport with the dispatcher or inside of an application, it would be convenient to have the transports logs go into the applications logs. Having convenient and pervasive logging for transports will make debugging transport failure easier.

Proposed solution

For each language-specific API, a language-specific logging mechanism will be added. The transport initializers will provide the means to pass information to the transports necessary for doing logging.

For the Go API, this proposal depends on the adoption of proposal 0013 - Migrate Go API from Factory Functions to a Factory Interface.

Design

1. Go Logging

In proposal 0013 - Migrate Go API from Factory Functions to a Factory Interface, a factory interface is introduced that supplies Dial() and Listen() functions. Instances of these interfaces are used both to create outgoing transport connections and to receive incoming transport connections. Creating instances of these interfaces is done by calling initializer functions that have flexibility in what parameters they take. Therefore, these initializer functions is where the logging capabilities will be added.

The signatures of the initializer functions for each transport can optionally take a log that can be used for logging. As the built-in Go standard library logging system does not support the required log levels, the open source logging package “go-logging” will be used. The full documentation for go-logging can be found here: https://github.com/op/go-logging

Examples of minimal logging-enabled initializer functions:

```
import "github.com/op/go-logging"

struct ExampleTransport {
      config string
      log *logging.Logger
}

func New(config string, log *logging.Logger) *ExampleTransport {
      return &ExampleTransport{config: config, log: log}
}
```

Adding a log parameter is optional in order to give transport developers flexibility in their transport implementation.

1.1 Using Go Logging from Inside a Transport

Import the logging library

```
import "github.com/op/go-logging"
```

The log has functions for each loglevel. The permitted logging functions are Debug, Info, Warning, and Error.

Examples of logging:

```
log.Debug("debug”)
log.Info("info")
log.Warning("warning")
log.Error("err")
```

Additionally, there are variant that allow logging a message which includes printf-style format strings and additional parameters. There is one such variant for each loglevel.

Example of format string logging:

```
log.Debugf("debug %s”, “example)
log.Infof("info %d", 100)
log.Warningf("warning, %s", “another example”)
log.Errorf("err %d", 500)
```

1.2 Setting up a Log in a Go Application

Applications must provide the log to the transport. The log can be configured with application-specific options.

Import the logging library

```
import "github.com/op/go-logging"
```

Create a log

```
var log = logging.MustGetLogger("example")
```

Optionally create a logging backend, otherwise the default backend will be used.

```
backend := logging.NewLogBackend(os.Stderr, "", 0)
```

This backend logs to stderr. No prefix is added to logging messages. No special flags are set.

Optionally, set the loglevel for the backend, otherwise the default loglevel will be used.

```
backendLeveled := logging.AddModuleLevel(backend)
backendLeveled.SetLevel(logging.ERROR, "")
```

This sets the backend’s loglevel to ERROR.

Finally, set the backend for the log.

```
log.SetBackend(backendLeveled)
```

2. Swift Logging

The signature of the factory initializer will be modified to optionally take a log that can be used for logging. As the built-in Swift standard library logging system does not support the required functionality, the open source logging package “swift-log” will be used. The full documentation for swift-log can be found here: https://github.com/apple/swift-log

Examples of a logging-enabled factory initializer:

```
class MeekConnectionFactory: ConnectionFactory
{
      init(to: URL, serverURL: URL, log: Logger)
}
```

Adding a log parameter is optional in order to maintain backwards compatibility.

2.1 Using Swift Logging from Inside a Transport

Add the logging library as a package dependency in your Package.swift file

```
.package(url: "https://github.com/apple/swift-log.git", from:
"1.0.0"),
```

Import the logging library

```
import Logging
```

The log has functions for each loglevel. The permitted logging functions are debug, info, warning, and error.

Examples of logging:

```
log.debug("debug”)
log.info("info")
log.warning("warning")
log.error("err")
```

2.2 Setting up a Log in a Swift Application

Applications must provide the log to the transport. The log can be configured with application-specific options.

Add the logging library as a package dependency in your Package.swift file

```
.package(url: "https://github.com/apple/swift-log.git", from:
"1.0.0"),
```

Import the logging library

```
import Logging
```

Create a log

```
let log = Logger(label: "com.example.ExampleApp.main")
```

Optionally set a logging backend, otherwise the default backend will be used.

```
LoggingSystem.bootstrap(StreamLogHandler.standardError)
```

This backend logs to stderr. Several other backends are linked from the swift-log documentation.

3. Java Logging

The signatures of the factory classes will be modified to optionally take a log that can be used for logging. The built-in standard library logging system java.util.Logging will be used. The full documentation for java.util.Logging can be found here: https://docs.oracle.com/javase/10/core/java-logging-overview.htm#JSCOR-GUID-B83B652C-17EA-48D9-93D2-563AE1FF8EDA

Examples of logging-enabled factory classes:

For classic Java networking,

```
import java.util.Logging

public class TransportSocketImplFactory extends SocketImplFactory
{
      private String config;
```

```
      private Logger logger;

      public TransportSocketImplFactory(String config, log Logger)
      {
            self.config = config;
            self.logger = logger;
      }

      public SocketImpl createSocketImpl()
      {
            return TransportSocketImpl(self.config, self.logger);
      }
}
```

For NIO Java networking,

```
import java.util.Logging;

public class TransportSelectionProvider extends SelectionProvider
{
      private String config;
      private Logger log;

      public TransportSelectionProvider(String config, Logger log)
      {
            self.config = config;
            self.log = log;
      }

      public DatagramChannel openDatagramChannel() throws IOException
      {
            return TransportDatagramChannel(self.config, self.log);
      }

      public DatagramChannel openDatagramChannel(ProtocolFamily
family)throws IOException
      {
            return TransportDatagramChannel(self.config, self.log,
family);
      }

      public ServerSocketChannel openServerSocketChannel() throws
IOException
```

```
	{
		return TransportServerSocketChannel(self.config,
self.log);
	}

	public SocketChannel openSocketChannel() throws IOException
	{
		return TransportSocketChannel(self.config, self.log);
	}
}
```

Adding a log parameter is optional in order to maintain backwards compatibility.

1.1. Using Java Logging from Inside a Transport

Import the logging library

```
import java.util.logging.*;
```

The log has functions for each loglevel. The permitted logging functions are severe, info, warning, and fine. The names for the loglevels in Java differ from the other language-specific logging APIs, but correspond to the same ERROR, INFO, WARNING, and DEBUG loglevels.

Examples of logging:

```
log.fine("debug”); // This corresponds to the DEBUG log level
log.info("info");
log.warning("warning");
log.severe("error"); // This corresponds to the ERROR log level
```

1.2 Setting up a Log in a Java Application

Applications must provide the log to the transport. The log can be configured with application-specific options.

Import the logging library

```
import java.util.logging.*;
```

Create a log

```
Logger logger = Logger.getLogger("com.example.App");
```

Optionally create a logging backend, otherwise the default backend will be used.

```
Handler handler = MemoryHandler();
```

This backend stores logs in memory.

Optionally, set the loglevel for the backend, otherwise the default loglevel will be used.

```
handler.setLevel(Level.INFO);
```

This sets the backend's loglevel to INFO.

Finally, set the backend for the log.

```
logger.addHandler(handler);
```

Effect on API Compatibility

These changes provide new capabilities to the APIs, but do not add any new requirements. Therefore they are backwards-compatible and require only a minor change to the specification.

Effect on IPC Compatibility

This proposal does not affect the IPC protocol. However, this proposal makes new functionality available in the API which could then be incorporated by another proposal into the IPC protocol.

Alternatives considered

There are two alternatives to this proposal. The first is to allow logging to continue the way it is currently done, with transports keeping their own logs in the transport state directory, or using unspecified logging APIs. The call from developers has been clear that integrating transport log messages into the application's log is preferable.

The other alternative to consider a different type of improved error handling. Logging is one way to achieve better debugging of failing transports. The other way to achieve better debugging is by a richer and more robust set of error values in the API. These two solutions represent two different approaches to debugging failures. Logging is an unstructured approach where they application can provide human-readable information about what went wrong, whereas error values are a machine-readable and human-readable way to provide information about failures which the application might be able to recover from, depending on the error. The goal of this

proposal is not to choose one approach over the other. Rather, both better logging and better error handling can both be incorporated under different proposals.

For the Go API specifically, an alternative logging solution was considered that didn't depend on proposal 013 - Migrate Go API from Factory Functions to a Factory Interface. However, this change would have not been backwards compatible. Since a new major version is required and both of these proposal address the same part of the API, it would be easier to adopt both at once.