Chapter 10
Middleware Case Study: MeDICi

Adam Wynne

10.1 MeDICi Background

In many application domains in science and engineering, data produced by sensors, instruments, and networks is naturally processed by software applications structured as a pipeline.[1] Pipelines comprise a sequence of software components that progressively process discrete units of data to produce a desired outcome. For example, in a Web crawler that is extracting semantics from text on Web sites, the first stage in the pipeline might be to remove all HTML tags to leave only the raw text of the document. The second step may parse the raw text to break it down into its constituent grammatical parts, such as nouns, verbs, and so on. Subsequent steps may look for names of people or places, interesting events or times so documents can be sequenced on a time line. Using Unix pipes, this might look something like this:[2]

```
curl47 http://sites.google.com/site/iangortonhome/ |    \
  totext | \
  parse | \
  people \   \
  places  -O out.txt
```

Each of these steps can be written as a specialized program that works in isolation with other steps in the pipeline.

In many applications, simple linear software pipelines are sufficient. However, more complex applications require topologies that contain forks and joins, creating pipelines comprising branches where parallel execution is desirable. It is also increasingly common for pipelines to process very large files or high volume data streams which impose end-to-end performance constraints. Additionally, processes in a pipeline may have specific execution requirements and hence need to be distributed as services across a heterogeneous computing and data management infrastructure.

From a software engineering perspective, these more complex pipelines become problematic to implement. While simple linear pipelines can be built using minimal

[1]http://en.wikipedia.org/wiki/Pipeline_%28software%29

[2]http://en.wikipedia.org/wiki/CURL

infrastructure such as scripting languages, complex topologies and large, high volume data processing requires suitable abstractions, run-time infrastructures, and development tools to construct pipelines with the desired qualities of service and flexibility to evolve to handle new requirements.

The above summarizes the reasons we created the MeDICi Integration Framework (MIF) that is designed for creating high-performance, scalable, and modifiable software pipelines. MIF exploits a low friction, robust, open-source middleware platform and extends it with component and service-based programmatic interfaces that make implementing complex pipelines simple. The MIF run-time automatically handles queues between pipeline elements in order to handle request bursts and automatically executes multiple instances of pipeline elements to increase pipeline throughput. Distributed pipeline elements are supported using a range of configurable communications protocols, and the MIF interfaces provide efficient mechanisms for moving data directly between two distributed pipeline elements.

The rest of this chapter describes MIF's features, shows examples of pipelines we've built, and gives instructions on how to download the technology.

10.2 MeDICi Hello World

We'll start with a description of the classic universal salutations example with MIF. The *helloWorld* sample in this section demonstrates a simple two-stage MIF pipeline. In order to understand this example, below are four simple definitions of MIF concepts that are used in the code.

1. *Pipeline*. A MIF application consists of a series of processing modules contained in a processing pipeline. The *MifPipeline* object is used to control the state of the pipeline, as well as to add and register all objects contained in the pipeline.
2. *Implementation code*. This is the code (e.g., a Java class or C program) that performs the actual processing at a given step in the pipeline.
3. *Module*. A module wraps the functionality of some implementation code with an interface that encapsulates the implementation. An instantiated module can be included as a step in a MIF pipeline.
4. *Endpoint*. Connects a module to other modules in the pipeline by using one of several standard communication protocols, such as JMS, SOAP, and many others.

In the MIF *helloWorld* example, when the pipeline starts, the user is prompted to enter a name. The name is sent to the *helloNameModule* which calls the *helloNameProcessor* implementation to add "Hey" to the front of the name and passes the whole string to the *helloHalModule*. This calls the *helloHalProcessor* to add "what are you doing" to the end of the string, which is returned to the user via the console.

In Fig. 10.1, the blue rectangles are modules and the black squares are endpoints. The path of data is represented by blue-dotted lines, which are annotated with the data strings that are passed between each step.

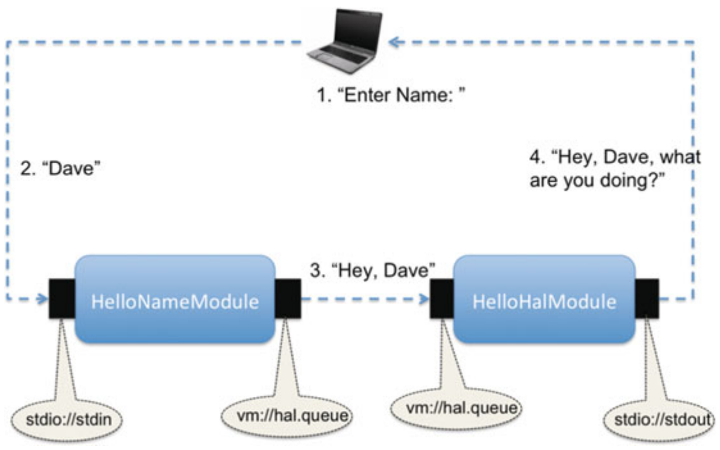


Fig. 10.1 MIF "Hello World" example

The following code snippets demonstrate the pertinent portions of the code. The code is presented in a "top-down" manner where we present the higher level code first and move progressively down to the implementation details.

First, we need to create a MifPipeline object which is needed to start and stop the processing pipeline. This object is also used to create and register all the objects that run within the pipeline.

```
MifPipeline pipeline = new MifPipeline();
```

Next, we add the *HelloNameModule* which prepends "Hey" to the entered name and sends the new string to the next stage in the pipeline. The first argument is a string representing the full class name of the implementation class. The second and third arguments are the inbound and outbound endpoints which allow our *HelloNameModule* to receive and send data.

```
pipeline.addMifModule(HelloNameProcessor.class.getName(),
"stdio://stdin?promptMessage=enter name: ","vm://hal.queue");
```

Lastly, we add the *HelloHalModule* (and its endpoints) to the pipeline. This calls the *HelloHalProcessor* to add another sentence fragment on the end of the string and prints it to the user's console.

```
pipeline.addMifModule(HelloHalProcessor.class.getName(),
"vm://hal.queue", "stdio://stdout");
```

Modules in the above example communicate using *endpoints*, which are passed to a module as arguments. Endpoints are an abstraction which enable the communication protocols between modules to be flexibly specified. This encapsulates the module implementation logic from having to be concerned with the communications protocols used to exchange messages with other modules. This means for example that a module configured with a JMS endpoint can be changed to send data over UDP without any change to the module's implementation code.

In this example, the *HelloNameModule's* inbound endpoint, *stdio://stdin*, reads user input from the console. That is, *stdio* is a special protocol that reads from the console and sends the console data to the module. The outbound endpoint, *vm://hal.queue*, is a MIF-provided endpoint implemented in the JVM, providing an efficient, queued communication mechanism between modules. Note that for modules to be able to communicate, the outbound endpoint of the sender must be the same as the inbound endpoint of the receiver.

After the pipeline modules are configured in the pipeline, we start the application by calling the method *MifPipeline.start().* This starts the MIF with the pipeline configuration and initiates the modules to listen for data.

```
pipeline.start();
```

Module implementation code is provided by the pipeline designer to perform some form of processing on the data flowing through a pipeline. This code is then wrapped by a module to form the smallest code unit which may be placed into a pipeline. The implementation code can be written in Java or any other language. When using Java, the code is integrated directly into the pipeline (it is treated as an external executable for other languages, as is explained later). For now, we will concentrate on creating Java implementation classes.

Implementation classes need to implement the *MifProcessor* interface, which provides a *listen* method with the following signature:

```
public Serializable listen(Serializable input);
```

This method is called when a message arrives for the module that wraps the implementation class. The input argument is the data received from the previous module in the pipeline, and the return value is the message which is sent to the next module in the pipeline.

Now let's take a look at the implementation of one of the modules from this example. The *HelloNameProcessor* implements the functionality for the *HelloNameModule*. When this module receives data, its *listen()* method is called, and the input data is passed in as the method argument. The method then processes the data in some way and returns the result that is passed on to the next module in the pipeline. In this case, the *listen* method simply adds the string "Hey" to the front of the received string and returns the new string.

```
public class HelloNameProcessor implements MifProcessor {
  public Serializable listen(Serializable name) {
    String str = "Hey, " + name;
    System.out.println("HelloNameProcessor: " + str);
    return str;
  }
}
```

10.3 Implementing Modules

A MIF *Module* represents the basic unit of work in a MIF pipeline. Every module has an implementation class, known as a MIF processor. The processor class is specified as the first argument to the *addMifModule* factory method when you create a module:

```
MifModule myModule =
pipeline.addMifModule(MyMifProcessor.class,"vm://in.endpoint"
, "vm://out.endpoint");
```

Each processor class must implement a *Processor* interface and an associated *listen* method that accepts an object representing the data payload received on the module's inbound endpoint. The *listen* method also returns an object representing the payload which is sent via the module's outbound endpoint.

```
public class MyMifProcessor implements MifObjectProcessor {
     public Object listen(Object input) {
     // perform some processing on input
       return output;
     }
}
```

There are a few different types of processor interfaces depending on whether you want to enforce the use of serialized objects and whether the processor needs to explicitly handle message properties that are sent as a header on all messages that pass through a MIF pipeline. These interfaces are found in the *package gov.pnnl. mif.user* and explained below:

10.3.1 MifProcessor

The *MifProcessor* interface is used to implement a module if you want to enforce that the types sent and received by the module are *Serializable*.

```
public interface MifProcessor {
  public Serializable listen(Serializable input);
}
```

10.3.2 MifObjectProcessor

This is the most general type of interface, allowing any type of object to be received by the listen method. It is often desirable to check the type of object received (with the *instanceof* operator) to ensure that it matches the correct derived type.

```
public interface MifObjectProcessor {
    public Object listen(Object input);
}
```

10.3.3 MifMessageProcessor

This type of processor is used when it is necessary to have access to the message properties associated with a given message. Normally, a processor receives just the message payload, but this interface allows the module to receive both.

```
public interface MifMessageProcessor {
    public Object listen(Object input,
                         MessageProperties messageProperties);
}
```

10.3.4 Module Properties

Since the processor class for a given *MifModule* is instantiated by the underlying MIF container, it is not possible to manually create and configure a processor class before adding it into a pipeline. Therefore, module properties are provided by the API to enable the user to pass any processor properties to MIF. MIF will then populate the properties on the processor when it is instantiated. The properties are set on the processor similarly to JavaBean properties, meaning that the class has a zero-argument constructor and standard setter and getter methods.

For example, the following is a *MifProcessor* with JavaBean-style setters:

```
public class Apple implements MifObjectProcessor {

    private String color;
    private String type;

    public Object listen(Object input) {
       // do stuff
       return output;
    }
    /* The apple's color */
    public setColor(String color) {
        this.color = color;
    }
    /* The type/variety of apple */
    public setType(String type) {
        this.type = type;
    }
}
```

The properties for this module can then be set with the following code:

```
MifModule  appleModule  =  pipeline.addMifModule(Apple.class,
"vm://in.endpoint", "vm://out.endpoint");
appleModule.setProperty("color", "red");
appleModule.setProperty("type", "Honeycrisp");
```

10.4 Endpoints and Transports

In MIF, communication between modules is enabled by transports, which are responsible for abstracting network communications and passing messages throughout a pipeline. Each communication protocol that is supported by MIF (e.g., JMS or HTTP) is implemented by a separate transport. Most of the complexity of a transport is hidden from the user by the use of configurable endpoints, which allow a module to be oblivious to the communication protocols it is using. However, it is sometimes necessary or desirable to configure the attributes of a transport. In such circumstances, there are APIs which enable the programmer to explicitly create and configure a connector.

Each transport has its own type of endpoint, which is used to connect modules to each other so that they can exchange messages. Each module has a set of inbound and outbound endpoints which can be set using the MIF API. To connect one module to another, the outbound endpoint of one module in the pipeline must match the inbound endpoint of another.

Endpoints are configured as strings representing a URI. It is possible to set properties on an endpoint to configure special behavior or override the default properties of a connector. For example, it is possible to configure whether an endpoint is synchronous or asynchronous by setting the "synchronous" property, as we'll explain soon.

An endpoint URI has the format:

```
scheme://host:port/path/to/service?property1=value1&property2
=value2
```

Where "scheme" is the particular type of transport being used (http, jms, vm, etc.); "host:port" is a hostname and port (which may or may not be present due to the nature of a given transport), and "path" is the path that distinguishes the endpoint from others. Properties are defined after a "?" at the end of a path and are delineated by key value pairs.

Each transport's endpoints are either synchronous or asynchronous by default. This default can be overridden by setting the "synchronous" property on an endpoint. For example, HTTP endpoints are synchronous by default. The following defines an HTTP inbound endpoint that creates an asynchronous service, listening at the address `localhost:9090/helloService`:

```
http://localhost:9090/helloService?synchronous=false
```

10.4.1 Connectors

Connectors are used to configure the attributes of a particular transport for the current pipeline or component. For example, the JMS connector allows the user to configure the location of the JMS server. Most of the time, the user does not need to explicity configure a transport since a default connector will automatically be created for each type of endpoint that occurs in the pipeline. It is however necessary to create and configure connectors when:

1. It is necessary to optimize the performance of a transport. This is common, for example, when using TCP endpoints.
2. No default connector can be created. For example, the use of JMS requires an explicit connector because it is necessary to specify the location of the JMS server.

If there is only one connector for a given protocol in a MIF pipeline, all endpoints associated with that transport will use this connector. However, multiple connectors may exist for the same transport in a single pipeline. In this case, you need to specify the name of the connector as a property on the endpoint so that MIF knows which connector to use for that particular endpoint.

For example, the following code excerpt shows a JMS connector defined with the name `jms-localhost`. Then, a module is configured with an inbound endpoint which specifies that connector, using the "connector" endpoint property. This allows endpoints in a MIF pipeline to connect to multiple JMS servers:

```
MifConnector conn = pipeline.addMifJmsConnector
  ("tcp://localhost:61616", JmsProvider.ACTIVEMQ);
conn.setName("jms-localhost");
MifModule fullNameModule = pipeline.addMifModule(
  NameArrayProcessor.class,
  "jms://topic:NameTopic?connector=jms-localhost",
  "stdio://stdout");
```

Properties can be set on a connector by calling the `setProperty` method on a `MifConnector`, e.g.:

```
MifConnector stdioConn =
  pipeine.addMifConnector(EndpointType.STDIO);
stdioConn.setProperty("messageDelayTime", 1000);
```

10.4.2 Supported Transports

The MIF API supports a number of transports. Below is a description of these, along with a description of the useful properties which can be set on endpoints and/or connectors of this type. All endpoints support the `connector=connectorName` property as described above.

10.4.2.1 VM

The `VM` endpoint is used for communication between components within the JVM. The key property that can be set on a `VM` endpoint is whether it is synchronous or asynchronous. If this property is set to `synchronous=true`, messages are passed synchronously from one module to another so that a slow receiver could slow down the sender. On the other hand, if this property is set to false, the sender sends as fast as it can without waiting for the receiver to complete each request. Internally, the MIF container manages a queue of messages associated with the connector.

For example, here is an example of an asynchronous `VM` endpoint

```
"vm://myqueue?synchronous=false"
```

10.4.2.2 STDIO

The `STDIO` transport is used to read from *standard in* and write to *standard out*. It is useful for testing and debugging. An example of `STDIO` endpoints is:

```
MifModule appleModule = pipeline.addMifModule(
   TextProc.class,
   "vm://in.endpoint",
   "vm://out.endpoint");
```

10.4.2.3 Java Messaging Service

The JMS transport connects MIF endpoints to JMS destinations (topics and queues). It is possible to use any JMS server provider with MIF. For convenience, the MIF installation includes ActiveMQ as the preferred JMS provider (and the provider MIF is tested with).

By default, JMS endpoints specify queues. In ActiveMQ, queues must be created administratively. For example, the following URI specifies that an endpoint connects to the queue called “aQueue”

```
jms://aQueue
```

To specify a topic, prepend the destination name with the string “topic:”. For example, the following URI specifies an endpoint connected to the topic called “bTopic”

```
jms://topic:bTopic
```

To create an ActiveMQ JMS connector, it is necessary to specify the server URI, as well as the JMS provider. Specifying the provider in this way allows provider-specific properties to be automatically set on the connector:

```
pipeline.addMifJmsConnector(
"tcp://localhost:61616",
JmsProvider.ACTIVEMQ);
```

10.4.2.4 HTTP

The HTTP transport enables inbound endpoints to act as Web servers and outbound endpoints to act as http clients. HTTP endpoints are synchronous by default. The following is an HTTP inbound endpoint that creates an asynchronous service, listening at the address `localhost:9090/helloService`.

```
http://localhost:9090/helloService?synchronous=false
```

10.4.2.5 HTTPS

The HTTPS transport enables the creation of secure services over HTTP. To create such a service, the connector must be configured to point to the *keystore* where the service's certificate is located. This requires the following properties to be set:

Connector properties	Description
keyStore	The location of the java keystore file
keyStorePassword	The password for the keystore
keyPassword	Password for the private key

As an example, the following MIF pipeline creates an HTTPS connector which is configured to use a self-signed certificate that can be created with the Java *keytool*.[3]

```
MifPipeline pipeline = new MifPipeline();

MifConnector httpsConn = pipeline.addMifConnector
(EndpointProtocol.HTTPS);
httpsConn.setProperty("keyStore", "/dev/ssl/keys/pnl.jks");
httpsConn.setProperty("keyStorePassword", "storepass");
httpsConn.setProperty("keyPassword", "keypass");

pipeline.addBridgeModule("https://hostname:9090/secureService
", "stdio://out");
pipeline.start();
```

10.4.2.6 TCP

This transport enables the creation of servers listening on raw TCP sockets (for inbound endpoints) and clients sending to sockets (outbound endpoints). As an example, the following endpoint URI will create a server listening on the given host and socket if used as an inbound endpoint.

```
tcp://localhost:7676
```

[3]http://download.oracle.com/javase/1.4.2/docs/tooldocs/windows/keytool.html

The following is an example of a TCP connector definition. Since the TCP protocol does not have the concept of a message, it is necessary to define a "protocol" algorithm and set this as a property on the connector. Thus, you should always explicitly create a TCP connector and choose an appropriate TCP processing protocol.

```
MifConnector conn = pipeline.addMifConnector
(EndpointProtocol.TCP);
conn.setProperty("tcpProtocol", new EOFProtocol());
```

The `EOFProtocol` class instantiated in this example is a Mule-provided class which defines the message boundary to be when EOF (end of file) on the socket is received. That is, the message ends when the client disconnects from the server by sending an EOF character. All of the available protocols can be found in Mule's TCP transport documentation.[4] For example, a protocol is provided that assumes each message is preceded by the number of bytes to be sent, so that an entire message can be constructed. It is also possible to specify a custom, application-specific protocol class.

Other properties can be set on TCP connectors to optimize performance. These include the size of the send and receive buffers, and for outbound endpoints, whether a socket should stay open after each send in order to improve throughput.

Since MIF is a specialization and simplification of Mule, the transports used in MIF are a subset of those in Mule. Thus, the documentation here is a highly condensed form of the Mule transport documentation. It may be useful to refer to the full Mule documentation to learn the full set of features provided by Mule.[5] The documentation focuses on the properties required by MIF applications.

10.5 MeDICi Example

The example application we describe here analyzes Internet chat messages to extract various types of content from the messages. The overall structure of the application is shown in Fig. 10.2.

Basically, when the application starts, the main program initializes a MIF pipeline and then simulates an external process that is pulling chat messages off of a network and inserting them into the pipeline via JMS. From there, the *Ingest* module takes a line of chat data and parses it into an object (`MapWrapper`) that is utilized throughout the rest of the pipeline. From the *Ingest* module, separate copies of the data (in the form of a `MapWrapper`) are routed to three concurrent processing modules (the actual logic of the processing is delegated to the

[4]http://www.mulesoft.org/documentation/display/MULE2USER/TCP+Transport

[5]http://www.mulesoft.org/documentation/login.action?os_destination=%2Fdisplay%2FMULE2USER%2FTCP%2BTransport

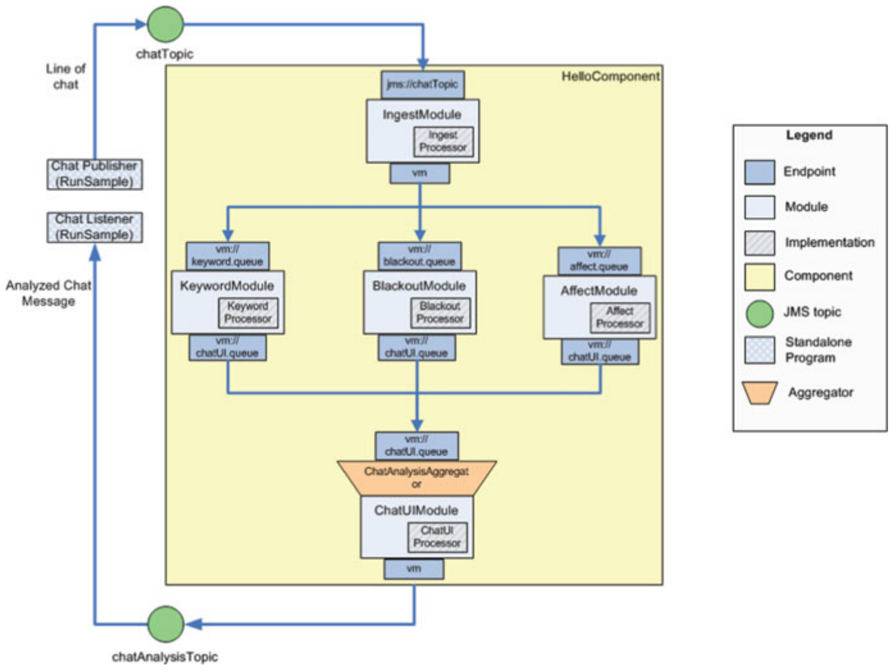


Fig. 10.2 MIF chat analysis pipeline

chat-specific code called by the MIF module and is beyond the scope of this description). Next, an aggregator combines the three resulting data objects into one message that is forwarded outside the pipeline for display, in this example's case, to the console.

Let's examine this pipeline in more detail.

10.5.1 Initialize Pipeline

First, we need to create a `MifPipeline` object which is needed to start and stop the processing pipeline. A `MifPipeline` is also used to create and register all the objects that run within the pipeline.

```
MifPipeline pipeline = new MifPipeline();
```

Next, we create and add a JMS connector, giving it the server address and the name of the server which it'll be using (in this case we use an *ActiveMQ* JMS provider).

```
pipeline.addMifJmsConnector
       ("tcp://localhost:61616", JmsProvider.ACTIVEMQ);
```

Next, we create the `ChatComponent` object, assign the endpoints, and add it to the pipeline. At this stage, we can start the pipeline so that it's ready to start receiving messages. As we'll see below, the heavy lifting of the pipeline configuration is encapsulated in the `ChatComponent` component.

```
ChatComponent chat = new ChatComponent();
chat.setInEndpoint("jms://topic:ChatDataTopic");
chat.setOutEndpoint
       ("stdio://stdio?outputMessage=CHAT RESULT: ");
pipeline.addMifComponent(chat);
pipeline.start();
```

Finally, we need to invoke a utility method that simulates a stream of chat messages flowing into the pipeline over JMS by reading a file of chat messages and sending them into the pipeline.

```
simulateChatStream();
```

10.5.2 Chat Component

The `ChatComponent` encapsulates the configuration of the application modules into an internal pipeline. First, we set the component endpoints that are passed in from the calling code (*ChatComponentDriver.java* in this case). Note how the component is oblivious to the transport that is associated with the endpoints, a JMS topic and *stdout* in this case. These details are abstracted completely in the component code, and hence the component can be used to communicate over any transport that is associated with its endpoints.

```
public void setInEndpoint (String inEndpoint) {
 this.inEndpoint = inEndpoint;
}
public void setOutEndpoint (String outEndpoint) {
 this.outEndpoint = outEndpoint;
 }
```

`ChatComponent` has a number of internal modules. First, `ingest-Module` is responsible for taking a chat message and parsing it into a data structure (`MapWrapper`) to be processed by all of the downstream processing modules. This module has three outbound endpoints since the outgoing message will be routed to three downstream processing modules (*Affect*, *Blackout*, and *Keyword*).

```
MifModule ingestModule = pipeline.addMifModule
        (Ingest.class.getName(),
        inEndpoint,
        "vm://ingest.keyword.queue");
// and add extra outbound endpoints
ingestModule.addOutboundEndpoint("vm://ingest.affect.queue");
ingestModule.addOutboundEndpoint("vm://ingest.blackout.queue");
```

Next, the downstream processing modules are connected by creating inbound endpoints that correspond to the `ingestModule` outbound endpoints (*ingest.keyword.queue* in the example below for the *Keyword* module (the others work similarly so we'll leave those out of this description)).

```
//Add KEYWORD Module
pipeline.addMifModule(Keyword.class.getName(),
         "vm:ingest.keyword.queue",
         "vm://keyword.queue");
```

Finally, the last step of the component configuration is to aggregate the results of the processing modules into one message and forward the result outside of the component using the outbound endpoint. To achieve this, we create the `chatAggregateModule` and connect to it the three outbound endpoints from the three upstream modules.

```
MifModule chatAggregateModule =
         pipeline.addMifModule(ChatAggregate.class.getName(),
         "vm://keyword.queue",
         outEndpoint);
chatAggregateModule.addInboundEndpoint("vm://affect.queue");
chatAggregateModule.addInboundEndpoing("vm://blackout.queue");
```

Aggregators are special MIF modules that combine messages from multiple sources into a single message. They can be used to collate the results of modules working in parallel (as in our chat example here) or to reduce a high volume of messages into a single object. To collate groups of messages, a correlation identifier must be added to each message. Events that should be aggregated have an identical correlation value, enabling the aggregator to combine them. Any MIF module can be associated with an aggregator that defines how multiple input messages are combined.

To create a MIF aggregator, we need to extend the `AbstractMif-Aggregator` abstract class. This requires implementing two methods, `shouldAggregateEvents` and `doAggregateEvents`. Both of these methods take a single `MifEventGroup` object that contains a list of objects that have been received on the aggregator's inbound endpoints.

The first method, `shouldAggregateEvents`, is called each time a new message is received by the aggregator on any endpoint. Its return value is a Boolean, representing whether or not that group of events contains a complete set that is ready to be aggregated into a single message. Typical actions performed by this method include counting the number of messages in the event group (for aggregating the results of a certain number of parallel processes) or looking for a particular message's presence (for digesting messages arriving over a certain period of time).

The second method, `doAggregateEvents`, is called on a group of messages whenever `shouldAggregateEvents` returns a true result for that group. It returns an object that represents the value of the objects in that group aggregated together. In an application distributing requests for airline fares, for instance, the return value might represent the lowest fair returned after a 30-s timeout message is received. In our example, the `ChatAnalysisAggregator` is responsible for combining the messages from the three upstream modules. This is accomplished by using a correlation value (i.e., a unique message identifier for each message that is assigned at the ingest stage) and combining these when all three messages have been received (one for each upstream module). In the MIF API, we simply create the aggregator and attach it to the module that it must be associated with.

```
MifAggregator chatAnalysisAggregator =
pipeline.addMifAggregator (new ChatAnalysisAggregator());
chatAggregateModule.setAggregator(chatAnalsysiAggregator);
```

10.5.3 Implementation code

For this example, all processing modules are written in Java. They wrap specific text processing libraries that perform the actual application logic. Below is an example of one of the processing modules. The `BlackoutProcessor`, which implements various identity protection algorithms, implements the functionality of the `BlackoutModule`. This represents a very common example of utilizing a wrapper class to call the "real" processing logic (usually in a library or jar) without needing to make any changes to the code to incorporate it in a MIF pipeline.

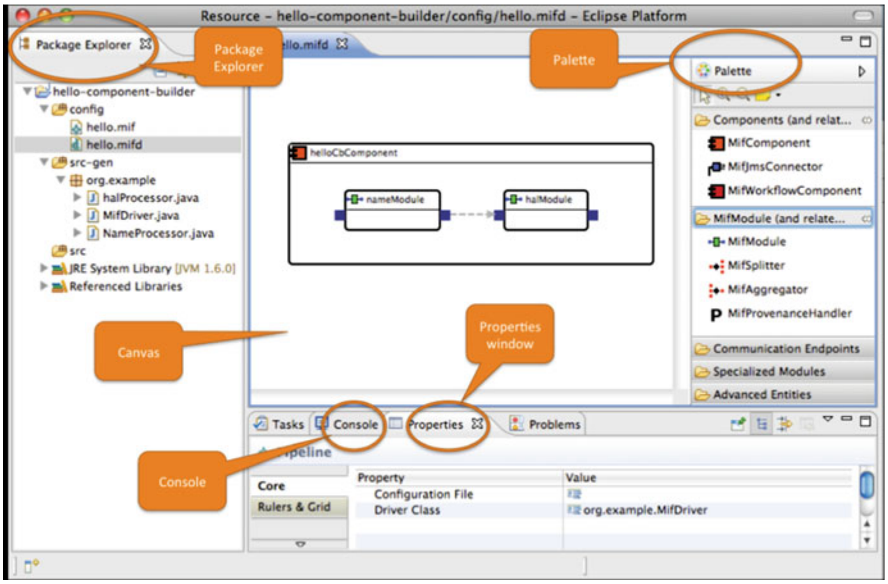
In this case, the code simply delegates to the application-specific method `blackout.processContentAnalysis(message)`.

```
blackout.processContentAnalysis(message).

public class BlackoutModule implements MifInOutProcessor {
  // lots of details omitted
  private static BlackoutId blackout = null;
  public BlackoutModule() {
    initBlackout();
  }
  public Serializable listen(Serializable input) {
    MapWrapper data = (MapWrapper) input;
    HashMap message = data.getMap();
    if(blackout != null){
      // call test processing logic
      blackout.processContentAnalysis(message);
    }
    return new MapWrapper(message);
  }
}
```

10.6 Component Builder

The MIF Component Builder (CB) is based on Eclipse and can be used by programmers to design MIF pipelines, generate stub code for objects in the pipeline, implement the stubbed code, and execute the resulting pipeline. The CB is capable



Property	Value
Configuration File	
Driver Class	org.example.MifDriver

Fig. 10.3 MIF component builder

of round trip development, supporting repeating the process of design, generate, implement, execute, until the programmer is satisfied with the pipeline. At that point, the components in the pipeline can be deployed to a running MIF instance or stored in a component library for later use. Figure 10.3 shows an example of using the CB and calls out the various windows that support development, namely:

- *Canvas*. The canvas represents the diagram file and is where the pipeline is configured. Objects are placed on the canvas and connected to create a pipeline. When objects are moved around the canvas and the diagram is saved, the changes are written to a *.mifd* file.
- *Palette*. The palette contains all the objects which make up a MIF pipeline model. The objects are selected from the palette and placed on the canvas. The palette separates objects into different categories for convenience. The *Components* section holds MIF components and commonly used objects which can appear inside a component. The *MifModule* section contains the *MifModule* object and other objects which can be placed inside a module. The *Communication Endpoint* section contains endpoint objects plus the *EndpointLink* which shows up on the canvas as a dotted line that connects an outbound endpoint on one module to the inbound endpoint on the next.
- *Package explorer*. The package explorer is used for organizing source files, configuration files, and MIF model files.
- *Properties window*. Properties for objects placed on the canvas are edited using the Eclipse "Properties" window. Selecting an entity in the canvas makes it possible to view and edit the necessary properties in the properties window. Properties for

the whole pipeline can also be set by clicking on the Eclipse canvas, and then by editing the properties that appear in the properties window. For example, note in Fig. 10.3 that the *Driver Class* property has the value *org.example.MifDriver*. This particular property means that this will be the class name of the generated driver class that runs the pipeline created in the component builder.

- *Console*. The console is just like the Java console in that it displays standard input and standard output within the IDE. In addition, the CB console also outputs status and results of code generation actions.

Each time a MIF design is saved, the underlying MIF model is checked for validity. This provides an error checking feature which is implemented by imposing constraints on objects in the model, such as "a *MifModule* must not have an undefined *implementationClass*." If such a constraint is not satisfied, the CB places a red "X" icon on the object(s) that are in error. When the user moves the mouse over one of these error icons, the CB provides a hint as to what the problem is.

10.7 Summary

We have used MIF in several applications over the last 3 years, in domains as diverse as bioinformatics, cybersecurity, climate modeling, and electrical power grid analysis. In all these projects, MIF has proven to be robust, lightweight, and highly flexible.

In building the MeDICi technology, we have been careful to leverage existing technologies whenever possible, and build as little software as possible to support the pipeline and component-based abstractions that our applications require. Part of the success of MeDICi therefore undoubtedly lies in the strengths of its foundations – Mule, ActiveMQ – which provide industrial-strength, widely deployed platforms. We see this as a sensible model for other projects to follow, especially in the scientific research community where resources for developing middleware class technologies are scarce.

10.8 Further Reading

The complete MeDICi project is open source and available for download from http://medici.pnl.gov. The site contains a considerable amount of documentation and several examples.

We've also written several papers describing the framework and the applications we've built. Some of these are listed below:

I. Gorton, H. Zhenyu, Y. Chen, B. Kalahar, B, S. Jin, D. Chavarria-Miranda, D. Baxter, J. Feo, *A High-Performance Hybrid Computing Approach to Massive Contingency Analysis in the Power Grid*, e-Science, 2009. e-Science '09. Fifth IEEE International Conference on e-Science, pp. 277–283, 9–11 Dec. 2009.

I. Gorton, A. Wynne, J. Almquist, J. Chatterton, *The MeDICi Integration Framework: A Platform for High Performance Data Streaming Applications*, wicsa, pp. 95–104, Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), 2008.

I. Gorton, Y. Liu, J. Yin, *Exploring Architecture Options for a Federated, Cloud-based Systems Biology Knowledgebase*, in 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2010) November 30 – December 3, Indiana University, USA, IEEE.