Engineering High Performance Service-Oriented Pipeline Applications with MeDICi

Ian Gorton, Adam Wynne, and Yan Liu

Pacific Nothwest National Lab
PO Box 999, Richland WA 99352, USA
{adam.wynne,yan.liu,ian.gorton}@pnl.gov

Abstract. The pipeline software architecture pattern is commonly used in many application domains to structure a software system. A pipeline comprises a sequence of processing steps that progressively transform data to some desired outputs. As pipeline-based systems are required to handle increasingly large volumes of data and provide high throughput services, simple scripting-based technologies that have traditionally been used for constructing pipelines do not scale. In this paper we describe the MeDICI Integration Framework (MIF), which is specifically designed for building flexible, efficient and scalable pipelines that exploit distributed services as elements of the pipeline. We explain the core runtime and development infrastructures that MIF provides, and demonstrate how MIF has been used in two complex applications to improve performance and modifiability.

Keywords: middleware, software pipelines, SOA, component-based systems.

1 Introduction

In many application domains in science and engineering, data produced by sensors, instruments and networks is commonly 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 geo-sciences, data on geology extracted from underground well drilling can be processed by a simple software pipeline that converts the raw data format to XML, extracts information from the files about drilling locations and depths, displays the location on a map-based interface and creates an entry in the database containing metadata for searching.

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 servicesacross 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

E.M. Maximilien et al. (Eds): ICSOC 2010 Workshops, LNCS 6568, pp. 88–99, 2011.

infrastructure such as scripting languages [2], 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.

In this paper we describe our 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.

MIF has been used at the Pacific Northwest National Laboratory (PNNL)in diverse application domains such as bioinformatics [3], scientific data management, cybersecurity, power grid simulation [4] and climate data processing [5]. In this paper, we briefly present the basic elements of MIF for building pipelines, and describe how these mechanisms can become the core of a SOA-based pipeline solution. We then describe the MIF Component Builder that provides a MIF design tool, and finally describe two case studies that demonstrate MIF's capabilities for text processing and cybersecurity.

2 Related Work

Various approaches exist for constructing software pipelines. The most common is to using scripting languages such as Perl, Python or shell scripts [7]. These work well for simple pipelines, as the lightweight infrastructure and familiar programming tools provide an effective development environment. Recently this approach has been extended in the Swift project [9], which has created a custom scripting language targeted at applications running on grid and high performance computing platforms. Swift has a number of attractive features for describing pipelines, but its implementation is limited in scope to large computational infrastructures which provide some necessary runtime infrastructure, for example, job scheduling.

Many workflow tools are also perfectly adequate for creating pipelines. Tools such as Kepler, Taverna and implementations of BPEL [6] can be used to visually constructapipeline-style workflow, link in existing components and services through a variety of protocols, including Web services, and then deploy and orchestrate the pipeline. These tools and approaches work well, but are heavyweight in terms of development infrastructure (ie they require visual development and custom workflow languages) and runtime overheads. The runtime overheads especially make them inappropriate for building pipelines that need to process high volume data streams, and small message payload sizes where fast context switching, lightweight concurrency and buffering are paramount.

Custom approaches also exist for building pipelines. Prominent in this category is Pipeline Pilot [1]. It provides a client-server architecture similar to that of most

BPEL-based tools. The client tools are used to visually create pipelines, in a custom graphical language, based upon underlying service-oriented components. The server executes pipelines defined by the client, orchestrating externally defined components through SOAP-based communications. The client also provides a proprietary scripting language to enable custom manipulation of data as it flows between the main steps of the pipeline.

The MeDICi Integration Framework (MIF) attempts to overcome the collective limitations of the above approaches by:

- allowing construction of pipelines using a standard, simple Java API
- providing a relatively lightweight runtime infrastructure (compared with other Java-based pipeline and workflow creation tools) suitable for both large data handling and long running computations, as well as bursty stream-based data with rapid, lightweight processing needs
- providing a strong separation of concerns between component behavior, inter-component communications and pipeline topology

3 MeDICi Integration Framework Overview

The MIF middleware platform is designed for building applications based on processing pipelines that integrate various software components using an asynchronous messaging platform. Figure 1 below shows a simple example of a processing pipeline with two analysis components.

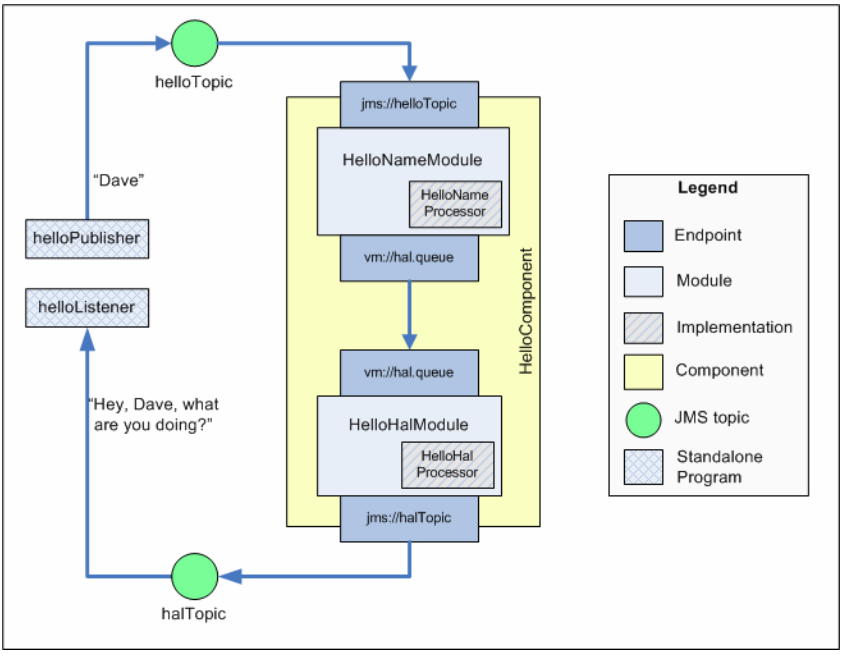

Fig. 1. A "Hello World" Pipeline in MIF

The key concepts in MIF API are:

- **MIF pipeline:** A processing pipeline is a collection of components connected as a directed graph. Data arrives at the first component in the pipeline, and the output of this component is passed to one or more downstream components for further processing. A pipeline can be of arbitrary length, components can accept inputs from one or more upstream components, and send their results to one or more downstream components.
- **MIF Module:** A MIF module is a user-supplied piece of code that inputs some defined data, processes it, and outputs some results. This code is wrapped by the MIF API to make it possible to plug the module in to a processing pipeline.
- **MIF Components:** MIF components provide a way to hierarchically compose MIF modules. A MIF component encapsulates a mini-pipeline of MIF modules.
- **Endpoints:** MIF modules define inbound and outbound endpoints as their connections to other modules in a pipeline. To connect two modules, the outbound endpoint from one module is configured to communicate with an inbound endpoint of another module.
- **MIF Container:** MIF components execute in the MIF container within a Java Virtual Machine. The container can automatically invoke multiple instantiations of pipeline component to handle multiple concurrent messages, and provides asynchronous, buffered communications mechanism to smooth out differences in the processing times of communicating components. This gives MIF pipelines the ability to seamlessly scale on multi-core platforms simply by adding more processing nodes and memory.
- **Local Components:** Local components are Java codes that are constructed using the MIF API. Local components execute within the MIF container.
- **Remote Components:** Remote components execute outside the MIF container, either on the same machine or on another node in a distributed application. The component code can be Java (e.g. an Message-Driven Bean), a Web service, HTTP server, a socket server or an executable written in any programming language. The MIF remote component API wraps the code and provides the capabilities to communicate with it from a MIF pipeline using a configurable protocol.
- **Messages:** Components pass messages down the pipeline as the incoming data is progressively transformed. Complete messages can be simply copied between components, or a reference to the message payload as a URIcan be exchanged to reduce the overheads of passing large messages.

Below is a simple example of configuring the MIF pipeline depicted in Figure 1.

```
// create the pipeline
MifPipeline pipeline = new MifPipeline();

// specify the JMS Connector
pipeline.addMifJmsConnector("tcp://localhost:61616",
JmsProvider.ACTIVEMQ);

//Add a component to the pipeline
HelloWorldComponent hello = new HelloWorldComponent();
pipeline.addMifComponent(hello);
```

```
// connect the component's endpoints and start processing
hello.setInNameEndp("jms://topic:NameTopic")
hello.setOutHalEndp("jms://topic:HalTopic")
pipeline.start();
```

The MIF container environment is provided by Mule[1], an open source messaging platform. For this reason, MIF shares many fundamental concepts with widely-used Enterprise Application Integration (EAI) and SOA-based integration brokers. Importantly for our implementation, Mule is lightweight, robust and scalable, traits which MIF inherits.

MIF extends the Mule API to make component-based pipeline construction simpler and to create an encapsulation mechanism for component creation. Our current implementation uses the ActiveMQ JMS for reliable message exchange, but the MIF API is designed as agnostic to the specific messaging platform. This allows deployments to configure MIF applications using JMS providers that meet their quality of service requirements.

4 MIF Component Builder

In order to simplify the creation of MIF pipelines, we created the MIF Component Builder to enable programmers to visually create and test pipelines inside of Eclipse Integrated Development Environment (IDE). Using this tool, programmers can visually create a pipeline, generate stub code for *MifModules* and *UserImplemented* entities, progressively implement the necessary code stubs, and run the pipeline inside the IDE. Thisprovides round trip development in which the programmer can cycle through design, generate and test until the pipelineis ready to be deployed.

We leveraged Model-driven Development (MDD) techniques in the Component Builder that enable the automated creation of model editors. Model-driven software development formalizes abstract software models as the basis for automated transformation into other models and/or programming code. This formalization allows for automated tools to operate on one or more models to generate applications. At the heart of these tools is the Eclipse Modeling Framework[2] (EMF). In addition to a modeling framework, EMF provides a code generation facility and runtime environment that allows models to be expressed in multiple formats and converted to a standard XML interchange (XMI) based format.

The MIF metamodel (a portion of which is shown in Figure2) is specified in EMF's ecore format. In the lower left of Figure2 is the *Pipeline*class, which is the root object of any MIF program. A MIF *Pipeline*comprises one or more *MifComponents*, which are high-level groupings of processing elements known as *modules*. *MifComponent* can be thought of as a sub-pipeline that contains one or more modules which all inherit from *BaseModule*. Modules are linked to each other using communication endpoints: *InboundEndpoint* allows data to be received by a module and *OutboundEndpoint* allows modules to send data to other modules. The metamodel is populated using the Graphical Modeling Framework (GMF), which provides

[1] http://www.mulesoft.org/
[2] http://www.eclipse.org/modeling/emf/

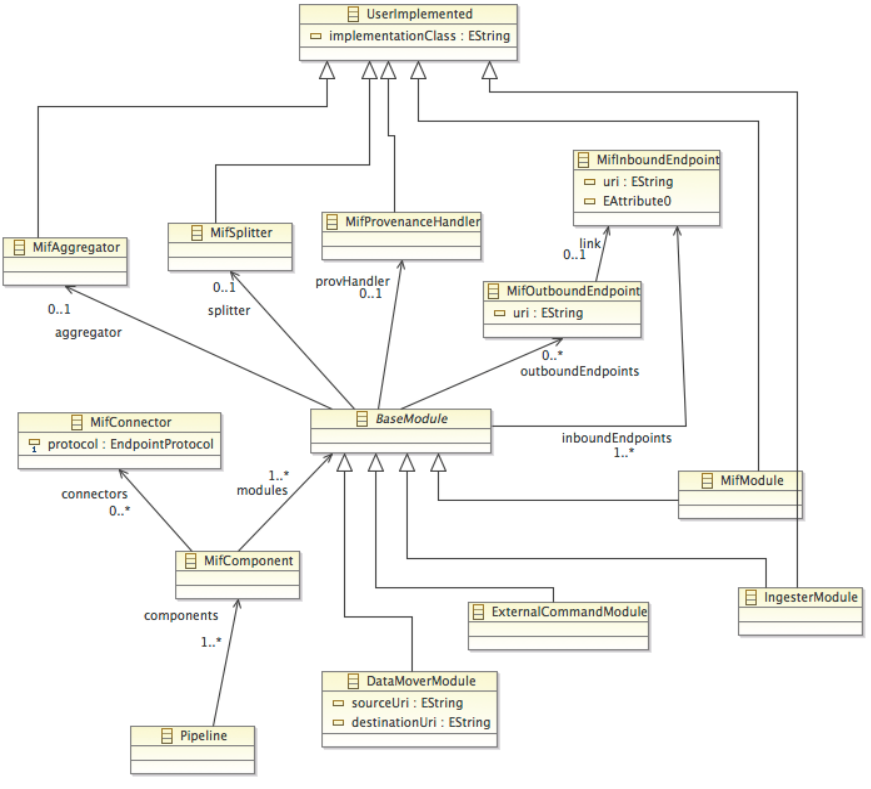

Fig. 2. MIF Base Component Model

model-driven tools for designing and generating graphical user interfaces. Once the metamodel is populated with the design of a particular MIF application, code generators produce Java that calls the MIF API. EMF therefore forms the basis for other tools that add capabilities for graphical editing, advanced template-based code generation, and model-to-model transformations, all of which are used to construct the MIF Component Builder.

5 Case Studies

5.1 Semantic Text Processing Pipeline

Performing semantic analysis of textual documents is a computationally intensive problem that is often structured as a pipeline. The first stage analyzes language structure, breaking down paragraphs and sentences into their constituent grammatical parts, and passes a marked-up version of the document to the next stage. Subsequent steps may for example identify places, events and people. In our application, the final stage compares the semantically marked up documents with an ontology and stores the relevant elements in a datastore as RDF triples.

The application was originally designed using the Apache UIMAtechnology[3]. The key UIMA component is an*annotator* that encapsulates text processing logic in a Java class. Multiple annotators providing specific text processing functions were built into a pipeline to handle each document. Each annotator produced a Java objectcontaining therelevant markups for processing by the downstream annotators. The output from one annotator is subsequently fed into the next annotator to discover more complex relations. Annotators are complex but can be designed independently and configured as needed for the specific application purposes.

The original system was developed as a monolithic UIMA application, as shown in Figure 3. The UIMA analysis engine (AE) is a container that hostsa pipeline of annotators. The configuration of these annotators is described by an XML file that specifies the detailed topology of the annotators, and inputs/outputs of each annotator. The UIMA AE loads the XML file and executes the pipeline of annotators.

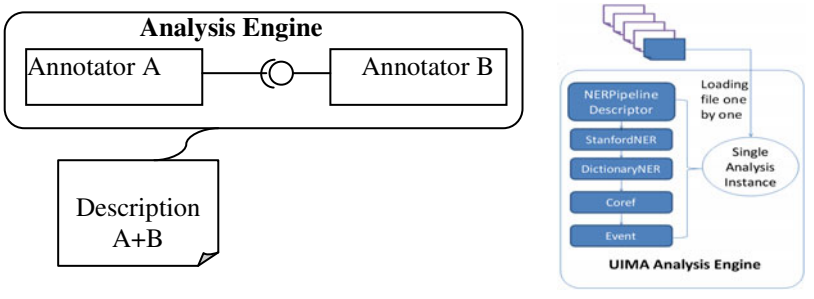

Fig. 3. Architecture of the UIMA pipeline

The monolithic UIMA architecture is straightforward to deploy on a single node, but UIMA has no native mechanism to expose the pipeline as a service.Further, it has several limitations in terms of performance and scaling. First, the pipeline is single-threaded, processing a single document at a time to completion. Hence parallel processing of documents requires multiple instances of the UIMA container to run concurrently. In addition, the processing time of each annotator on a single file was diverse. Two of the 5 annotators in the pipeline consumed 98% of the runtime. As a result, this tightly coupled architecture wascumbersome and heavyweight to scale to process tens of thousands of documents, especially as asingle document can take hundreds of seconds to emerge from the pipeline.

Hence an architectural solution was essentialto decouple the annotators so that optimization strategies such as introducing concurrency to individual annotators could be achieved.This required refactoring the original pipeline into separate annotation services that could be connected through MIF endpoints, as shown in Figure 4. AEs are deployed one separate nodes and host one or more annotation services. For

[3] `http://uima.apache.org/`

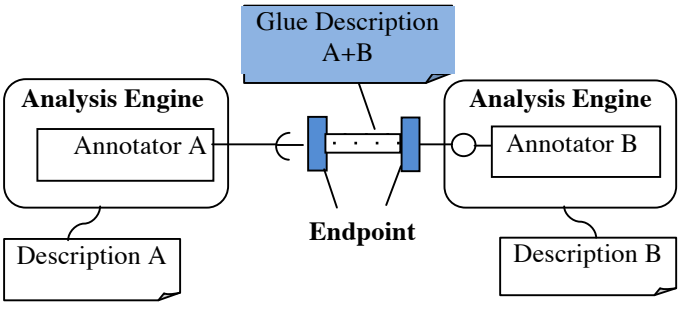

Fig. 4. Service Oriented Annotators Architecture

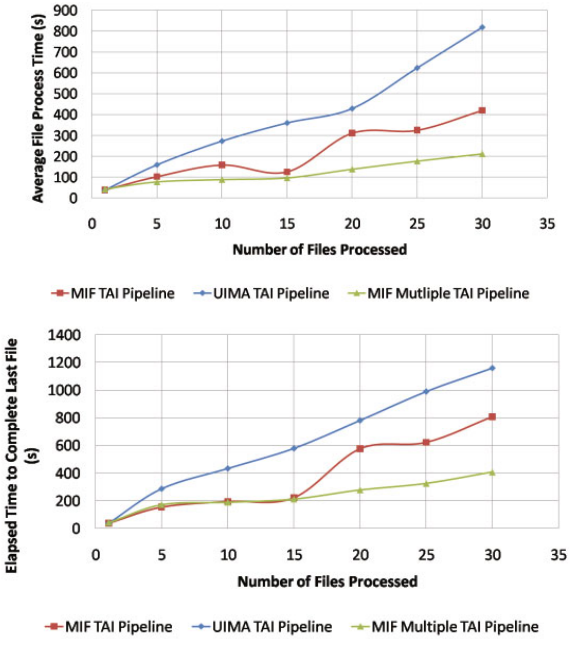

Fig. 5. Performance of threealternative architectures

example, the most time consuming annotator is deployed on one AE, while several other annotators demanding less processing time are bundled into another AE.

The challenge of implementing such an SOA is to compose annotation services into a pipeline that preserves the original analysis properties. We achieve this using MIF, which provides the essential mechanisms to glue together the distributed annotation services. The MIF solution wrapsanAE as a MIF component, and configures the components into a pipeline, which is exposed as a Web service.

In our solution, we devised two architecture alternatives that implement different refactoring strategies using MIF. The alternatives were:

1. A single MIF pipeline containing a MIF file ingester and MIF components wrapping individual annotators. The concurrency required for processing multiple documents simultaneously in the pipeline is managed by MIF by default. As a result, concurrent documents are efficiently processed by the MIF pipeline.
2. Three MIF pipelines are constructed - two for annotators dominating the document processing times, and one for the remainder. This creates a partitioned pipeline architecture. Each MIF pipeline is responsible for the internal concurrency of its MIF components, and the pipelines are connected by JMS endpoints.

We deployed the original architecture and the two refactored architectures using MIF on the same computing environment. We measured individualdocument processing times and the elapsed time to complete the last file in a batch. Figure 5shows the performance as the load of documents increases.

The results demonstrate that the refactored MIF pipelines improve performance significantly. The multiple MIF pipeline architecture scales linearly as the load increases and produces stable performance compared to the single MIF pipeline architecture. The reason is that multiple MIF pipelines balance the annotator workload better through more efficient resource utilization, giving significant performance improvements of up to a factor of 4.

5.2 Cyber Analytic Pipeline

The analysis of network data has traditionally been done in a forensic fashion in which firewall, host, and router logs(referred to as network flows) are stored for post-mortem searching in response to a suspected attack.Purpose built scripts, command-line tools, and graphical tools are manually used to comb through massive amounts of log activity. The workflow of this activity is a manual pipeline in which a set of files are retrieved and transformed from different sources into a common format. Next, programs are used to filter out data that represents normal traffic and find data items that resemble a certain traffic pattern or contain a set of IP addresses of interest.

This workflow presents several challenges, including: (1) repeating successful pipeline runs is challenging due to the number of ad hoc steps taken, (2) swapping out different transformation, analytics, and visualization codes is labor intensive and error prone, and (3) it does not scale to the network security problems encountered today in which data from entire networks must be rapidly aggregated and processed. Automated Intrusion Detection Systems (IDS) such as Snort[4]exist to alleviate some of the manual analysis involved in this pipeline. However, these are typically signature based and suffer from known problems of a high rate of false positives and the inability to detect attacks that are not known in advance.

Therefore, we have used MIF to construct a cyber analytics pipelinedue to its ease of swapping in and out components, message types, and network protocols. Further,

[4] http://www.snort.org/

MIF's support for the graphical creation, execution, and modification of pipelines greatly eases the burden of building robust and reusable pipeline applications.This makes it easy to sendanalytic outputs to multiple complementary visualization tools that are needed in order to detect complex, emerging, and novel network attacks [8].

Our MIF-based cyber analysis pipeline is able to consume, transform, filter and analyze network flow data as it is produced in real time. The pipeline generates data for three different visualization tools for real time analysis.

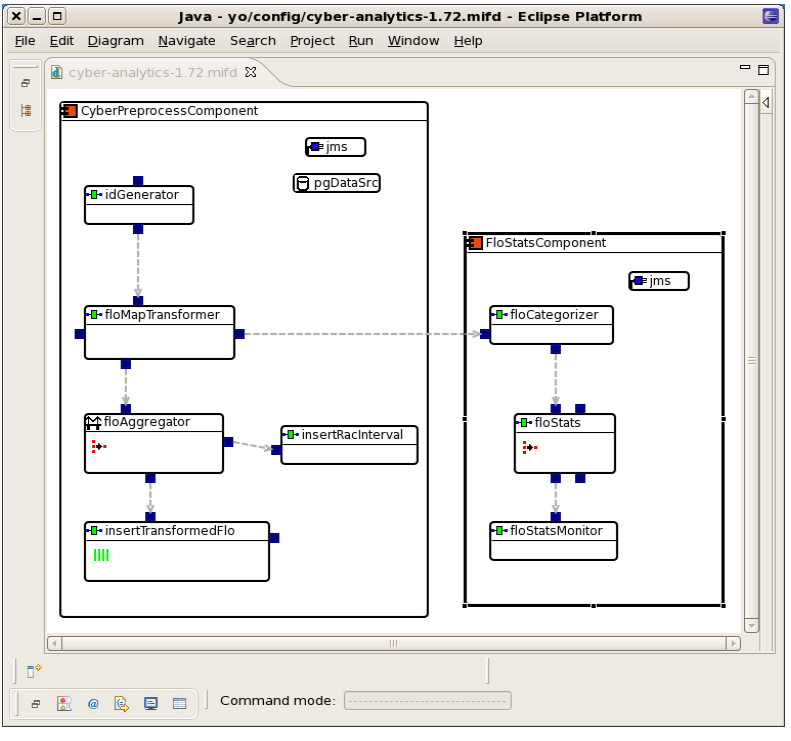

Fig. 6. Cyber pipeline design in MIF Component Builder

The pipeline, depicted in the MIF Component Builder in Figure6, comprises two high level components, namely *CyberPreprocessComponent* and *FloStatsComponent*. An external data ingest program (not shown) pushes data from network sensors over a JMS topic endpoint to the pipeline. *CyberPreprocessComponent* performs transformation, aggregation and loading of raw and aggregated data into the database, and outputs the data over another JMS topic for visualization displays to consume. The data is also sent to the *FloStatsComponent*, which categorizes the network data and attempts to fit the traffic patterns to one or more statistical distributions. These summary statistics are also sent over JMS for listening visualization tools to consume. If any incoming data is found to be anomalous, it is sent over a dedicated *event* topic, which triggers an alarm state in the visualization displays.

Performance tests were performed with MIF, the ActiveMQ JMS server, and a Postgres database. The MIF hostwas a dual quad core 2.80 GHz Intel Xeon® processor with 16 GB of memory.In testing with replayed network sensor inputs, the pipeline easily kept up with the average actual arrival rate of 83 flow msg/s and the peak of 145 msg/sWe then stress-tested the pipeline by progressively increasing the traffic replay rate well past real values. For the best run, the system reported a maximum average throughput of 2,781 msg/s, or over 240 million messages per day on a single node. The peak throughput 30-second interval was 3,350 msg/s. During this time, neither of the servers reached above 50% CPU utilization, leading us to conclude that communication between MIF components and the database wasthe bottleneck, and not the MIF pipeline execution.

6 Further Work and Conclusions

We are using MIF on several large projects in PNNL focusing on large-scale scientific data management and processing data from data streams. The technology is proving robust, fast and scalable. In addition, we are investigating using MIF as the core pipelining infrastructure for a petascale, federated collection of biology data. In this context, our work is focused on introducing adaptivity into MIF, so that based on policies or historical performance data, MIF pipelines can dynamically reconfigure, moving processing components to remote sites where the required data is located. Such a capability would enable us to more efficiently implement pipelines which process very large-scale data sets.

The complete MIF implementation is open source and freely available from http://medici.pnl.gov

References

1. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns, vol. 1. Wiley, Chichester (2009)
2. Yang, X., Bruin, R.P., Dove, M.T.: Developing an End-to-End Scientific Workflow. Computing in Science and Engineering, 52–61 (May/June 2010)
3. Shah, A.R., Singhal, M., Gibson, T.D., Sivaramakrishnan, C., Waters, K.M., Gorton, I.: An extensible, scalable architecture for managing bioinformatics data and analysis. In: IEEE 4th International Conference on e-Science, Indianapolis, Indiana, December 7-12, pp. 190–197. IEEE Computer Society, Los Alamitos (2008)
4. Gorton, I., Huang, Z., Chen, Y., Kalahar, B., Jin, S., Chavarria-Miranda, D., Baxter, D., Feo, J.: A High-Performance Hybrid Computing Approach to Massive Contingency Analysis in the Power Grid. In: Fifth IEEE International Conference on e-Science and Grid Computing, pp. 277–283. IEEE, Los Alamitos (2009)
5. Chase, J.M., Gorton, I., Sivaramakrishnan, C., Almquist, J.P., Wynne, A.S., Chin, G., Critchlow, T.J.: Kepler + MeDICi - Service-Oriented Scientific Workflow Applications. In: 2009 IEEE Congress on Services - Part I (Services-1 2009), pp. 275–282. IEEE, Los Alamitos (2009)

6. Barker, A., van Hemert, J.: Scientific Workflow: A Survey and Research Directions. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 746–753. Springer, Heidelberg (2008)
7. Kiebel, G.R., Auberry, K.J., Jaitly, N., Clark, D., Monroe, M.E., Peterson, E.S., Tolic, N., Anderson, G.A., Smith, R.D.: PRISM: A Data Management System for High-Throughput Proteomics. Proteomics 6(6), 1783–1790 (2006)
8. Best, D.M., Bohn, S., Love, D., Wynne, A., Pike, W.A.: Real-time visualization of network behaviors for situational awareness. In: Proceedings of the Seventh international Symposium on Visualization For Cyber Security, VizSec 2010, Ottawa, Ontario, Canada, September 14, pp. 79–90. ACM, New York (2010)
9. Wilde, M., Foster, I., Iskra, K., Beckman, P., Zhang, Z., Espinosa, A., Hategan, M., Clifford, B., Raicu, I.: Parallel Scripting for Applications at the Petascale and Beyond. Computer 42(11) (2009)