The RDF Pipeline Framework: Automating Distributed, Dependency-Driven Data Pipelines

David Booth

Independent Consultant
KnowMED, Inc.
david@dbooth.org
http://dbooth.org/2013/dils/pipeline/,
http://rdfpipeline.org/

Abstract. Semantic web technology is well suited for large-scale information integration problems such as those in healthcare involving multiple diverse data sources and sinks, each with its own data format, vocabulary and information requirements. The resulting data production processes often require a number of steps that must be repeated when source data changes -- often wastefully if only certain portions of the data changed. This paper explains how distributed healthcare data production processes can be conveniently defined in RDF as executable dependency graphs, using the RDF Pipeline Framework. Nodes in the graph can perform arbitrary processing and are cached automatically, thus avoiding unnecessary data regeneration. The framework is loosely coupled, using native protocols for efficient node-to-node communication when possible, while falling back to RESTful HTTP when necessary. It is data and programming language agnostic, using framework-supplied wrappers to allow pipeline developers to use their favorite languages and tools for node-specific processing.

Keywords: Data flow, data pipelines, semantic web, RDF, SPARQL.

1 Introduction

A major use case for semantic web technology in industry is information integration involving several diverse data sources, each having its own access protocols, data format, vocabulary and information content. Healthcare data fits this profile well. When semantic web technology is used for this purpose, source data such as patient information and lab data must be accessed, converted to RDF[1], and transformed in ways that are specific to each data source, to link the information together. Ontologies and rules are useful in performing semantic transformation of the information, and often require multiple processing steps. In addition, if the information is important – such as healthcare information – there are often multiple applications that must consume that information, i.e., multiple data *sinks,* each one having its own data format, vocabulary, information requirements and protocol requirements. For example, the same source information may be used for patient care

C.J.O. Baker, G. Butler, and I. Jurisica (Eds.): DILS 2013, LNBI 7970, pp. 54–68, 2013.
© Springer-Verlag Berlin Heidelberg 2013

purposes, research, quality-of-care measurement, billing, etc. This further complicates the data production process with more custom steps.

To automate the data production process when using semantic web technology, often an ad hoc pipeline is built using a mixture of shell scripts, SQL queries, SPARQL updates, web services, etc., and sometimes specialized integration tools. The resulting pipeline often uses a mix of interfaces ranging from files, to web services, HTTP, SQL, etc., and deals with a mixture of data representations such as text, CSV, XML and relational. On the plus side, such pipelines can be built using whatever tools are available for addressing each part of the problem, and the pipeline can evolve organically. On the minus side, such pipelines become extremely fragile, difficult to understand and difficult to maintain, both because they use so many technologies and because the topology of the pipeline is very hard for a newcomer to figure out. Typically, the topology is not expressed explicitly in one document – unless someone manually documented the pipeline, in which case the documentation is likely out of date. Instead, the topology is implicit in the communication that occurs between a shell script on one server, another shell script on another server, a web service on yet another server, etc. Furthermore, requirements frequently change as new data sources and new applications are integrated, thus causing pipeline maintenance to be a major problem.

To simplify the creation and maintenance of automated data production pipelines, various pipeline languages, tools and frameworks have been created over the years. For example, much research has already been done on workflow automation[2], and the W3C in 2010 standardized an XML pipe processing model, XProc[3]. Although the work presented in this paper could be considered workflow automation, it differs from most work in that area (and XProc) in that: (a) it is specifically oriented toward semantic web data production pipelines; and (b) it is more primitive, as there is no flow of control, no flow of control operators, and no central controller. A few other frameworks have been developed specifically for semantic web data production[4][5][6][7], but our work differs from those in being fully decentralized, with no central controller.

This paper presents an approach for semantic web data production pipelines that is unique in being decentralized – there is no central controller – distributed, web oriented (based on RESTful[8][9] HTTP), dependency graph driven, and allows adjacent nodes in a pipeline to transparently use local data-access methods when the nodes are compatible and on the same server. The approach was designed for semantic web applications but can also be used for other purposes. The approach has been implemented in the **RDF Pipeline Framework[10], an open source project available under the Apache 2.0 license**. The Framework provides: (a) a hosting environment (initially Apache2 using mod_perl2) with a pluggable wrapper interface; and (b) some standard wrappers such as FileNode, GraphNode and JavaNode. (Wrappers are discussed in Section 3.1.) The user provides: (a) an RDF pipeline definition (such Pipeline #1 shown below); and (b) updaters (described below). As of this writing (4-May-2013), code for the RDF Pipeline Framework is in "developer

release" status: it runs and passes regression tests, and may be downloaded for testing, but is not yet ready for general production release, as some code cleanup and documentation still need to be done.

2 Example Pipeline

To provide a concrete basis for illustrating this approach, this section presents a simple example of a data pipeline using the RDF Pipeline Framework. Figure 1 shows a data pipeline (Pipeline 1) for producing cardiology and immunology data based on patient medical records and lab data. There is no special significance to the names of the nodes in this pipeline (patients, labs, normalize, merge, process, cardiology, immunology). They were chosen only to suggest the application-specific processing that they might perform.

To keep the example very simple, only two data sources are used and they are both text files, though one comes from a remote HTTP source and the other from a local file. (Of course, an actual system would likely involve more data sources and the data sources would often be things like relational databases or web services.)

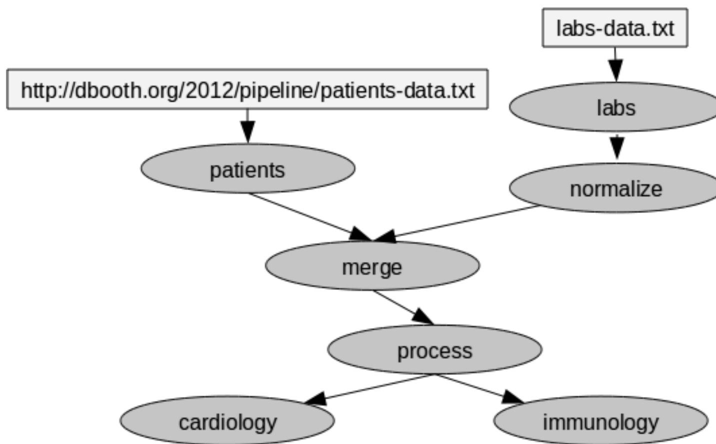

Fig. 1. This simple data pipeline (**Pipeline 1**) shows patient and lab data being combined to produce data that is consumed for cardiology and immunology purposes. Each node in the graph performs arbitrary application-specific processing and data storage. A directed link from one node to another indicates data flow and hence data dependency. Although the lab data is related to the patient data – lab results for patients – the lab data is first run through a "normalize" step before being merged with the patient data. After merging, the data is further processed through another application specific step before being consumed by the cardiology and immunology nodes.

Here is the content from **http://dbooth.org/2012/patients-data.txt**:

```
patient id=001  name=Alice      dob=1979-01-23
patient id=002  name=Bob        dob=1950-12-21
patient id=003  name=Carol      dob=1944-06-12
```

```
patient id=004  name=Doug       dob=1949-08-27
patient id=005  name=Ellen      dob=1966-09-29
patient id=006  name=Frank      dob=1971-11-15
```

And here is the content of file **labs-data.txt**:

```
lab     customer=001    glucose=75      date=2012-02-01
lab     customer=002    glucose=85      date=2012-02-02
lab     customer=002    glucose=94      date=2012-02-03
lab     customer=004    glucose=72      date=2012-03-01
lab     customer=004    glucose=104     date=2012-03-02
lab     customer=004    glucose=95      date=2012-03-03
lab     customer=005    glucose=98      date=2012-02-02
lab     customer=006    glucose=87      date=2012-01-15
lab     customer=006    glucose=91      date=2012-01-16
```

The pipeline of Figure 1 (Pipeline 1) is defined in RDF/Turtle[11] as follows. Line numbers have been added for reference purposes.

```
 1. # Pipeline 1: RDF/Turtle for Figure 1
 2. @prefix p: <http://purl.org/pipeline/ont#> .
 3. @prefix : <http://localhost/node/> .
 4.
 5. :patients a p:FileNode ;
 6.   p:inputs ( <http://dbooth.org/2012/patients-data.txt> ) .
 7.
 8. :labs a p:FileNode ;
 9.   p:inputs ( "labs-data.txt" ) .
10.
11. :normalize a p:FileNode ;
12.   p:inputs ( :labs ) .
13.
14. :merge a p:FileNode ;
15.   p:inputs ( :patients :normalize ) .
16.
17. :process a p:FileNode ;
18.   p:inputs ( :merge ) .
19.
20. :cardiology a p:FileNode ;
21.   p:inputs ( :process ) .
22.
23. :immunology a p:FileNode ;
24.   p:inputs ( :process ) .
```

It is easy to see that this pipeline definition corresponds directly to the graphical representation in Figure 1. Indeed, although Figure 1 was drawn manually, tools such as TopBraid Composer[12] can automatically display graphical representations of these pipelines, making them very easy to visualize. Some notes:

Line 2: Prefix "p:" is declared for the namespace <http://purl.org/pipeline/ont#> of the RDF Pipeline Framework's vocabulary. This is the vocabulary used to define a pipeline in the RDF Pipeline Framework, as summarized in Section 3.7.

Line 3: Prefix ":" is declared for the base URI <http://localhost/node/> of the RDF Pipeline server that will host one or more nodes in the pipeline. Any number of servers may be used, though this example uses only one.
Line 5: Node <http://localhost/node/patients> (abbreviated as :patients) is defined to be of type p:FileNode, which is the kind of wrapper (see Section 3.1) to be used by the :patients node. In web style, a node's URI is used both to identify that node and to retrieve data from it.
Line 6: The :patients node takes its input from a remote source, <http://dbooth.org/2012/pipeline/patients-data.txt>.
Lines 11-12: The :normalize node takes the result of the :labs node as its input.
Lines 14-15: The :merge node has two inputs, specified as an ordered list: the :patients node and the :normalize node.

Although Pipeline 1 defines the data flow between nodes, it supplies no details about the application-specific processing that is performed by each node. This separation of concerns makes it easy to reconfigure the pipeline without affecting the application-specific processing, and vice versa.

To specify the application-specific processing that a node should perform, an *updater* must be supplied. An ***updater*** is a named function, command or other operation that implements the processing task of a node. An updater is written by the user to perform an application-specific operation that produces data. Its job is to produce the node's output when invoked by its wrapper (described in Section 3.1). The wrapper passes, to the updater, wrapper-specific parameters for the node's inputs and output destination, such as filenames for a FileNode, or RDF graph names for a GraphNode.

For a node of type p:FileNode, such as :patients, the updater must be an executable program that accepts files as inputs and writes its output to stdout or (optionally) to a file. By default, the framework expects the name of the updater to be the node name implicitly, but it may also be specified explicitly using the p:updater property. Below is the updater for the :patients node, written as a shell script. Again, the line numbers are not a part of the script.

```
1. #! /bin/sh
2. # This is the patients node updater.
3. cat $1 | ./patients2rdf
```

This updater simply pipes the content of file $1 through ./patients2rdf and writes the result to stdout. Significant things to notice:

- There are no Application Programmer Interface (API) calls to pollute the updater code. Instead, the RDF Pipeline Framework invokes the updater when the data for that node needs to be generated, allowing the updater to be clean, simple and focused only on the application-specific task that it needs to perform.
- The updater expects its input as a file whose name is passed in as a parameter **$1** to the script, even though the pipeline definition specified its input as

<http://dbooth.org/2012/patients-data.txt>. The RDF Pipeline Framework will automatically cache – in a file – the content retrieved from http://dbooth.org/2012/patients-data.txt, and provide the cache filename as the actual parameter $1 when it invokes the updater.

As shown in line 15 of Pipeline 1, the :merge node expects two inputs – the :patients node and the :normalize node. Here is the :merge updater.

```
1. #! /bin/sh
2. # This is the merge node updater.
3. cat $1
4. cat $2 | sed 's/customer/patient/g'
```

The :merge updater performs a crude RDF merge by concatenating files $1 and $2 to stdout. It also performs some crude ontology alignment by filtering file $2 through *sed* in the process, to change all occurrences of "customer" to "patient", because the :labs data used the word "customer" where the :patients data used the word "patient". (Warning: this technique of using *cat* and *sed* to merge and edit RDF data will only work for certain kinds of data, and should not be used in general. It is shown here only to keep the example short and simple.)

Because the inputs of the :merge node are specified as an ordered list in the pipeline definition, parameter $1 of the :merge node updater corresponds to the output of the :patients node, and parameter $2 corresponds to the output of the :normalize node.

Once deployed, each node in a pipeline is independently "live", and will respond to data requests by dereferencing the node's URI. Thus, there are no specially designated endpoints: any node can be used as an endpoint or as an intermediate node. For example, if the :patients node is dereferenced – such as by pasting its URI into a browser, or by using the *curl*[13] command – the updater program named *patients* will be invoked (if necessary) and its output will be returned. Here is the output of "**curl http://localhost/node/patients**", with XSD data types[14] omitted for brevity:

```
@prefix patient: <http://example/patient#> .
@prefix : <http://example/med#> .
patient:p001 :lab [ :name "Alice" ; :dob "1979-01-23" ] .
patient:p002 :lab [ :name "Bob" ; :dob "1950-12-21" ] .
patient:p003 :lab [ :name "Carol" ; :dob "1944-06-12" ] .
patient:p004 :lab [ :name "Doug" ; :dob "1949-08-27" ] .
patient:p005 :lab [ :name "Ellen" ; :dob "1966-09-29" ] .
patient:p006 :lab [ :name "Frank" ; :dob "1971-11-15" ] .
```

And here is the output of "**curl http://localhost/node/merge**":

```
@prefix patient: <http://example/patient#> .
@prefix : <http://example/med#> .
patient:p001 :lab [ :name "Alice" ; :dob "1979-01-23" ] .
patient:p002 :lab [ :name "Bob" ; :dob "1950-12-21" ] .
```

```
patient:p003 :lab [ :name "Carol" ; :dob "1944-06-12" ] .
patient:p004 :lab [ :name "Doug" ; :dob "1949-08-27" ] .
patient:p005 :lab [ :name "Ellen" ; :dob "1966-09-29" ] .
patient:p006 :lab [ :name "Frank" ; :dob "1971-11-15" ] .
@prefix patient: <http://example/patient#> .
@prefix : <http://example/med#> .
patient:p001 :lab [ :glucose 750 ; :date "2012-02-01" ] .
patient:p002 :lab [ :glucose 850 ; :date "2012-02-02" ] .
patient:p002 :lab [ :glucose 940 ; :date "2012-02-03" ] .
patient:p004 :lab [ :glucose 720 ; :date "2012-03-01" ] .
patient:p004 :lab [ :glucose 1040 ; :date "2012-03-02" ] .
patient:p004 :lab [ :glucose 950 ; :date "2012-03-03" ] .
patient:p005 :lab [ :glucose 980 ; :date "2012-02-02" ] .
patient:p006 :lab [ :glucose 870 ; :date "2012-01-15" ] .
patient:p006 :lab [ :glucose 910 ; :date "2012-01-16" ] .
```

This technique of making each node independently "live" means that no central controller is needed or used, though nodes in a pipeline do share the same pipeline *definition*. It also allows the pipeline to be used for multiple applications that share some, but not all of the same data requirements. For example, the pipeline may have originally been built to supply an application with data from only the :merge node. The :cardiology and :immunology nodes may have been added later for other applications, without duplicating work or disrupting the existing pipeline. Furthermore, since each node can be on a different server (if desired), accessing its own private data, nodes can run concurrently.

The RDF Pipeline Framework does not currently check to see if a pipeline contains a cycle, although such a check would be straight-forward to add using well-known techniques. Since a pipeline definition indicates data dependencies, a cycle would likely be a mistake, though it is conceivable that a use could be found for it.

3 The RDF Pipeline Approach: What It Does and How It Works

This section describes more of the principles used in this approach, how they work and how they are used.

3.1 Wrappers

Pipeline 1 above showed how an updater could be implemented by an arbitrary executable program, such as a shell script, which took files as inputs and produced a file as output. However, although shell scripts and files are convenient in many cases, data preparation for semantic web applications often requires processing steps that are more conveniently and efficiently performed directly within an RDF data store. Approaches like this are convenient for transforming RDF data from one model, ontology or vocabulary to another. For example, SPARQL 1.1 Update[15] operations can be used to create RDF named graphs from other named graphs. One can consider such tasks to be nodes in a pipeline, in which SPARQL Update operations take named graphs as inputs and produce named graphs as outputs.

To accommodate such needs, a node is composed of two parts: the updater and a *wrapper*. A ***wrapper*** is a standard component, usually provided by the Framework, that is responsible for invoking the updater and communicating with other nodes. This architecture allows updaters to be written in any programming language and consume or produce any kind of object, provided that a suitable wrapper is available. A wrapper runs inside a ***hosting environment*** that implements an HTTP server (e.g., Apache2/mod_perl2 or Tomcat), allowing the wrappers to respond to HTTP requests, and in turn potentially invoking updaters. The wrapper framework is extensible, so new wrapper types can be plugged in to each hosting environment. The wrapper must be implemented in the same programming language as its hosting environment (e.g., Perl or Java), but this does not necessarily need to be the same language in which updaters are written – it depends on the wrapper.

Some basic wrappers:

- **p:FileNode**, for updaters written as executable programs (in any programming language) that consume and produce files;
- **p:GraphNode**, for updaters written as SPARQL Update operations that consume and produce RDF named graphs in a SPARQL server; and
- **p:JavaNode**, for updaters written in Java that consume and produce Java objects in a JVM.

For example, the following SPARQL Update code INSERTs presidents from graph http://example/in to graph http://example/out whose foaf:givenName is "Bill", changing the foaf:givenName to "William". Again, the line numbers are not a part of the code.

```
 1. # SPARQL Updater #1
 2. PREFIX foaf:     <http://xmlns.com/foaf/0.1/>
 3. PREFIX inGraph:  <http://example/in>
 4. PREFIX outGraph: <http://example/out>
 5.
 6. DROP SILENT GRAPH outGraph: ;
 7.
 8. INSERT {
 9.   GRAPH outGraph: {
10.     ?president foaf:givenName "William" .
11.     ?president foaf:familyName ?familyName .
12.     }
13.   }
14. WHERE {
15.   GRAPH inGraph: {
16.     ?president foaf:givenName "Bill" .
17.     ?president foaf:familyName ?familyName .
18.     }
19.   }
```

Unfortunately, although the above code could be used as a p:GraphNode updater, it would not be very convenient or flexible, because the names of the input and output graphs are hard coded. Thus, the code would need to be modified if the pipeline were reconfigured to use a different input or output graph. It would be nice if the graph

names were instead passed in as parameters, so that this same SPARQL code could be used on any input and output graphs, but SPARQL 1.1 does not provide any way to do that. The RDF Pipeline Framework therefore includes a simple template facility that can be used for this purpose. Here is the same updater code, but written as a SPARQL Update template.

```
 1. # SPARQL Updater #2, using a template
 2. #inputs ( ${in} )
 3. #outputs ( ${out} )
 4.
 5. PREFIX foaf:     <http://xmlns.com/foaf/0.1/>
 6. PREFIX inGraph:  <${in}>
 7. PREFIX outGraph: <${out}>
 8.
 9. DROP SILENT GRAPH outGraph: ;
10.
11. INSERT {
12.   GRAPH outGraph: {
13.     ?president foaf:givenName "William" .
14.     ?president foaf:familyName ?familyName .
15.     }
16.   }
17. WHERE {
18.   GRAPH inGraph: {
19.     ?president foaf:givenName "Bill" .
20.     ?president foaf:familyName ?familyName .
21.     }
22.   }
```

Points worth noting:

Line 1 is a normal SPARQL comment line.

Line 2 tells the SPARQL template processor the names of this updater's formal input parameters. When the template is expanded at runtime, this line will be removed and every occurrence of ${in} will be changed to the URI of the node's input.

Line 3 tells the SPARQL template processor the names of this updater's formal output parameters. When the template is expanded at runtime, this line will be removed and every occurrence of ${out} will be changed to the node's URI.

3.2 Serializing, Deserializing and Optimizing Communication

In addition to invoking a node's updater, the wrapper is responsible for communication between nodes. Thus, the wrapper performs wrapper-specific serialization of node data (such as serializing a graph to RDF/Turtle) when it needs to transmit that data to an external node or other requester, and it performs the corresponding deserialization upon receiving data from an external node. This allows updaters to stay very simple – unpolluted by serialization, deserialization or data transmission issues.

This wrapper architecture also allows adjacent nodes to communicate more efficiently when they are on the same server and use the same wrapper. Instead of serializing an object, transmitting it via HTTP and deserializing it on receipt, nodes in the same environment can transparently access each other's objects directly. For example, if node <http://example/in> were an input to node <http://example/out> in a pipeline, and both nodes were p:GraphNodes in the same server, then the updater for <http://example/out> would automatically directly access the graph produced by <http://example/in>, avoiding both HTTP and serialization / deserialization.

Pipelines can be built from a heterogeneous mix of node types as long as the serializations produced by the wrappers are compatible. For example, a p:FileNode could produce output that is RDF/Turtle and be used as the input of a p:GraphNode.

3.3 Caching and Updating Only When Necessary

A wrapper does not necessarily invoke a node's updater for every data request. The wrapper automatically caches a node's output and keeps track of whether any of the node's inputs have changed. The updater is invoked only if the cached output is stale with respect to the nodes inputs. Again, this allows updaters stay simple, focusing only on the application-specific tasks that they need to perform.

3.4 Deploying and Distributed Processing

As of this writing, a pipeline is deployed by placing the pipeline definition file and updaters into the deployment directory of each hosting environment and starting the hosting environments, such as Apache2. However, a future version of the Framework will likely allow the pipeline definition to be read from an arbitrary HTTP source, thus simplifying the distribution of a new version of the pipeline definition to multiple hosting environments.

Nodes in a pipeline can be deployed on any servers that are accessible to their adjacent nodes. Consider the following simple two-node pipeline.

```
1. # Pipeline 2
2. @prefix p: <http://purl.org/pipeline/ont#> .
3. @prefix b: <http://server1.example.com/> .
4. @prefix w: <http://server1.example.com/> .
5. b:bills a p:GraphNode ;
6.   p:inputs ( <http://dbooth.org/2012/presidents.ttl> ) .
7. w:williams a p:GraphNode ;
8.   p:inputs ( b:bills ) .
```

Lines 3 and 4 of Pipeline 2 indicate that the b:bills and w:williams graphs are actually in the same SPARQL server (server1.example.com), and thus the p:GraphNode wrapper will cause the w:williams node to access b:bills graph directly. In contrast, if we had deployed these nodes on different servers (server1.example.com and server2.example.com) the pipeline definition would differ only on line 4, as shown in Pipeline 3 below. Furthermore, the updaters would not change at all.

```
1. # Pipeline 3
2. @prefix p: <http://purl.org/pipeline/ont#> .
3. @prefix b: <http://server1.example.com/> .
4. @prefix w: <http://server2.example.com/> .
5. b:bills a p:GraphNode ;
6.   p:inputs ( <http://dbooth.org/2012/presidents.ttl> ) .
7. w:williams a p:GraphNode ;
8.   p:inputs ( b:bills ) .
```

3.5 Update Policies

Consider Pipeline 3 above, and suppose that the data from node b:bills changes. When should the updater of node w:williams be invoked to update its output? Should it be updated immediately? Or should it be updated only when its output is actually requested? Or perhaps periodically, every *n* seconds?

A node's p:updatePolicy may be specified as an additional node property to indicate the policy that the wrapper should use in deciding when to invoke a node's updater. Potential policies include p:lazy, p:eager and p:periodic – each one identifying a particular algorithm that will be used internally. Again, by specifying the update policy in the pipeline definition, a node's updater can stay simple.

3.6 Passing Parameters Upstream

Pipeline 1 above showed :cardiology and :immunology both consuming data. However, each one may only need a small subset of the total data that is available. It would be wasteful to propagate all possible :patients and :labs data through the pipeline if only a small subset is actually needed. For example, cardiology may only need data for patient=(002,003,004), and immunology may only need data for patient=(003,006).

To avoid this problem, parameters can be passed upstream through the pipeline, as illustrated in Figure 2. By default such parameters are passed as query string parameters on a node's URI, when node data is requested. For example, the command "curl 'http://localhost/node/cardiology?patient=(002,003,004)'" will request data from node :cardiology, passing query string "patient=(002,003,004)" as a parameter. (Of course, if parameters contain sensitive information then they should be suitably encrypted.) A parameter is treated as an additional node input, and thus a parameter change can cause the node's updater to fire. A node's updater can make use of its parameters if it chooses to do so. For example, for a p:FileNode updater, the most recently passed parameter is available in the $QUERY_STRING environment variable, and the parameters from all of a node's output nodes are available in the $QUERY_STRINGS environment variable.

By default, parameters are propagated upstream automatically. However, a pipeline definition may specify a p:parametersFilter for any node in order to transform the parameters as they are propagated upstream through that node. A p:parametersFilter is thus analogous to an updater, but it only operates on parameters

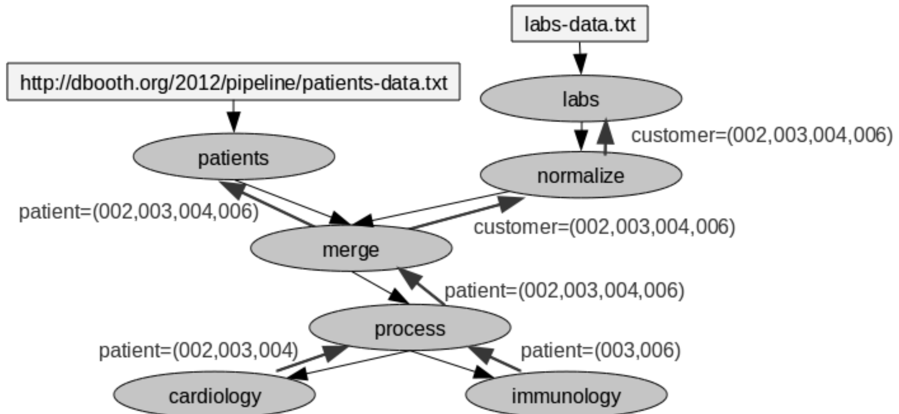

Fig. 2. Parameters are passed upstream through the pipeline, to control the data that the :patients and :labs nodes will generate. By default, parameters are passed upstream without modification. However, the pipeline definition may specify a p:parametersFilter for a node to control how that node will combine and/or modify the parameters that it passes upstream. In this illustration, the :process node supplied a p:parametersFilter that merged the parameters "patient=(002,003,004)" and "patient=(003,006)" that it received from :cardiology and :immunology, to produce the parameter "patient=(002,003,004,006)" that it passes upstream to the :merge node. The :merge node then used another p:parametersFilter to pass one parameter "patient=(002,003,004,006)" upstream to the :patients node, but a different parameter "customer=(002,003,004,006)" to the :normalize node. No other node in this pipeline needs to specify a p:parametersFilter. By passing such parameters upstream, the :patients and :labs updaters are able to generate only the data that is actually needed downstream.

that are being passed upstream. For a p:FileNode, the p:parametersFilter must be an executable program – typically a simple shell script. This treatment of parameter propagation again allows updaters to stay simple, while providing a powerful technique for data production to be efficiently controlled.

3.7 Error Checking and Automated Transformations

As of this writing, the Framework provides minimal error checking and does not include monitoring or alerting functions, though such features could be added in a future version. The Framework itself would be useful in implementing such features. For example, it would be easy to write an email notification node that reads from an error stream.

The Framework knows almost nothing about the semantics of a node or its inputs or output. It does not check to ensure that the actual input that a node receives conforms to the media type that the node expects, nor does the Framework perform any automatic transformation from one media type to another. It would be straightforward to extend the Framework to add such error detection and/or automatic transformation, but this has not been done thus far, because: (a) the user would have

to declare the expected media types for each of a node's inputs, thus making the pipeline definition more verbose; (b) the correspondence between the pipeline definition and the actual processing would be less direct, since in essence the Framework would perform automatic translation by inserting implicit translation nodes into the pipeline as needed; (c) a node normally has input expectations that go far beyond what a media type specifies, and during development these expectations need to be tested anyway, to ensure that the node receives what it expects, so it seems quite unlikely that a media type mismatch would pass unnoticed during such testing; and (d) it is very easy to insert an explicit translation node into a pipeline anyway.

Since an updater can perform arbitrary processing, updaters can have side effects that are unknown (and unknowable) to the Framework. Such side effects could cause concurrency issues if different updaters share the same resource. Users should bear this in mind when designing their updaters.

3.8 Graceful Evolution of Nodes and Pipelines

One motivation for cleanly separating the application-specific concerns (encapsulated in a pipeline's updaters) from the mechanics of caching, updater invocation, serialization, deserialization and handling HTTP requests, is to enable nodes and pipelines to evolve gracefully, without impacting other part of the pipeline: loose coupling. For example, a node can be swapped out for a new version, implemented in an entirely different programming language, with no change to adjacent nodes and only a trivial change to the pipeline definition (to change the node's wrapper type). This enables a pipeline to be developed quickly and easily, using the simplest available updater implementation techniques, and then refined as needed, adding features or improving efficiency. This fits well with agile development practices.

3.9 RDF Pipeline Properties

Section 3.1 discussed wrappers, which are represented in a pipeline description as classes. The following table summarizes the user-oriented properties used in defining a pipeline. Wrappers use additional properties internally. The subject (or domain) of each property in the table is a node unless the Value column indicates otherwise, such as "Subject is $nodeType", which means that the subject should be the *type* of a node, e.g., GraphNode, rather than a node instance. For all properties (and classes) the namespace is <http://purl.org/pipeline/ont#> except for the rdfs:type property, a/k/a "a" in Turtle.

Property	**Value**
a / rdfs:type	Node type, e.g., GraphNode.
contentType	HTTP Content-Type for this node's serialized output. Defaults to defaultContentType of the $nodeType.
defaultContentType	Subject is $nodeType. Default HTTP Content-Type for serialized output.

defaultContentEncoding	Subject is $nodeType. Default HTTP Content-Encoding for serialized output.
dependsOn	URIs of inputs, parameters and anything else this node depends on. Inputs and parameters are automatically included, but dependsOn can be used to specify additional dependencies.
hostRoot	Subject is $nodeType. The value is a list that maps the server prefix (such as "http://localhost") of node URIs of this $nodeType to the root location (as native name) of the server that implements the wrapper for this $nodeType. Analogous to $DOCUMENT_ROOT, which is used by default if this property is not set. Example: p:GraphNode p:hostRoot ("http://localhost" "http://localhost:28080/openrdf-workbench/repositories/owlimlite/") .
inputs	URIs of this node's inputs. They maybe other RDF Pipeline Nodes, or arbitrary HTTP data sources.
parametersFilter	File path of parametersFilter, relative to server "$ENV{DOCUMENT_ROOT}/node/".
state	Native name of node output, i.e., the object that will be updated by the node's updater. For example, for a FileNode it is a filename. For a GraphNode it is a named graph.
stateType	Subject is $nodeType. Type of state, if set. Otherwise $nodeType is used.
stderr	File name of stderr from last update.
updatePolicy	Specifies the name of the algorithm that decides whether/when a node's state should be updated. Potential policies include lazy, eager and periodic.
updater	Native name of updater function.

4 Security

Data security is critical in healthcare and many other domains. For lack of space, this paper does not detail how security concerns can be addressed in the RDF Pipeline Framework, but as a brief outline:

- Wrappers can ensure that data in transit is securely encrypted, both in passing data downstream and in passing parameters upstream.
- Secure HTTP (https:) can also be used, for an additional layer of inter-node communication security.
- Updaters can ensure that data at rest is fully encrypted, if necessary.

5 Conclusions

This paper has presented a novel approach to automating data production pipelines for healthcare and other applications using semantic web technology. The approach makes use of framework-supplied *wrappers* that handle caching, dependency checking and inter-node communication, allowing a node's updater code to stay simple and application-focused. This also allows the framework to be used with multiple programming languages or object types, given appropriate wrappers. The approach is decentralized – every node in a pipeline is live – and nodes can be easily distributed across multiple servers with minimal change to the pipeline definition and no change to a node's updater. The approach is implemented as an open source project at http://rdfpipeline.org/ . Interested parties are invited to contact the author.

References

1. W3C: Resource Description Framework (RDF), `http://www.w3.org/RDF/` (retrieved June 08, 2012)
2. Anonymous, Articles on workflow (google scholar search), `http://tinyurl.com/a5yf5ng` (retrieved February 01, 2013)
3. Walsh, N., Milowski, A., Thompson, H., XProc: An XML Pipeline Language, W3C Recommendation (May 11, 2010), `http://www.w3.org/TR/xproc/` (retrieved February 01, 2013)
4. Becker, C., Bizer, C., Isele, R., Matteini, A., et al: Linked Data Integration Framework (LDIF), `http://www4.wiwiss.fu-berlin.de/bizer/ldif/` (retrieved June 08, 2012)
5. Top Quadrant: Sparql Motion, `http://www.topquadrant.com/products/SPARQLMotion.html` (retrieved June 08, 2012)
6. Phuoc, D.L., Morbidoni, C., Polleres, A., Samwald, M., Fuller, R., Tummarello, G.: DERI Pipes, `http://pipes.deri.org/` (retrieved June 08, 2012)
7. Fensel, D., van Harmelen, F., Witbrock, M., Carpentier, A.: LarKC: The Large Knowledge Collider, `http://www.larkc.eu/` (retrieved February 01, 2013)
8. Methedras: REST for the Rest of Us, `http://developer.mindtouch.com/REST/REST_for_the_Rest_of_Us` (retrieved June 08, 2012)
9. Fielding, R.: Chapter 5: Representational State Transfer (REST). From PhD Thesis: Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine (2000), `http://roy.gbiv.com/pubs/dissertation/rest_arch_style.htm` (retrieved June 08, 2012)
10. Booth, D.: rdf-pipeline, A framework for RDF data production pipelines, google code repository, `http://rdfpipeline.org/` (retrieved February 01, 2013)
11. Prud'hommeaux, E., Carothers, G. (eds.): Turtle: Terse RDF Triple Language (2011), `http://www.w3.org/TR/turtle/` (retrieved June 08, 2012)
12. TopQuadrant: TopBraid Composer, `http://www.topquadrant.com/products/TB_Composer.html` (retrieved June 08, 2012)
13. Stenberg, D.: curl man page, `http://curl.haxx.se/docs/manpage.html` (retrieved June 08, 2012)
14. Biron, P.V., Malhotra, A.: XML Schema Part 2: Datatypes Second Edition (2004), `http://www.w3.org/TR/xmlschema-2/` (retrieved June 08, 2012)
15. Gearon, P., Passant, A., Polleres, A.: SPARQL 1.1 Update (2012), `http://www.w3.org/TR/sparql11-update/` (retrieved June 08, 2012)