Corpora Camp Platform Architecture and API

The scholarly Activity Groupings can be divided up into the following broad categories:

- Modeling
  - Discover (Searching)
  - Object Management (Creating, Updating, Deleting)
  - Sourcing/Citing (Reading, Referencing)
- Controlling/Computing
  - Analysis (from Model to View)
  - Machine-mediated Coordination (from View to Model)
- Viewing
  - Presentation
  - Output

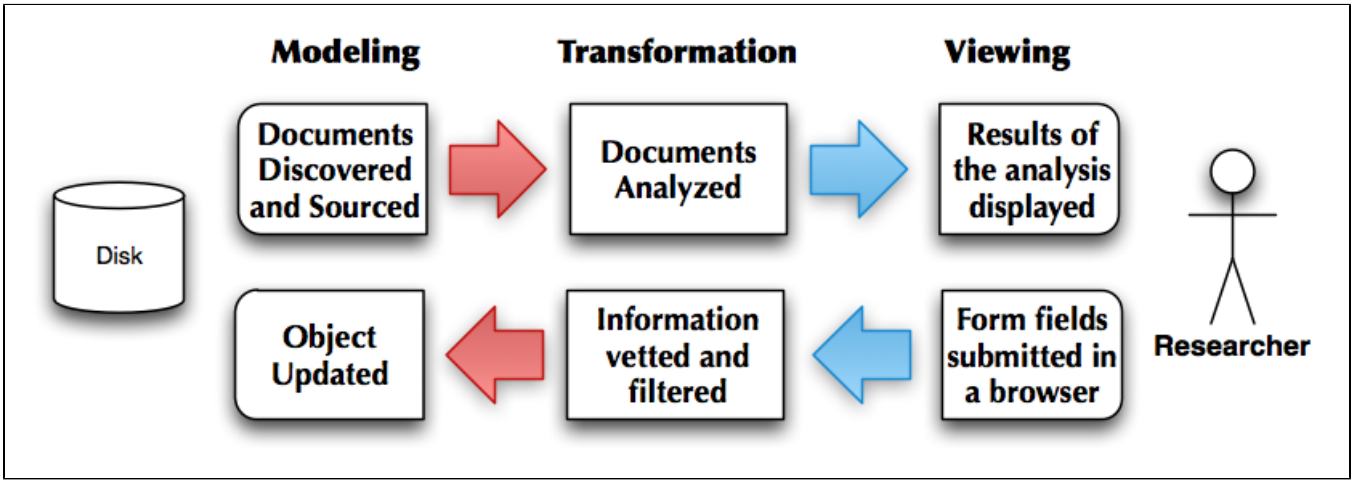
Underlying these three can be a framework enabling Preservation. Augmenting the framework can be an Identity manager allowing, for example, annotation of Objects with owner and author attribution. Corpora Camp isn't concerned with Preservation or Identity, so we won't provide those capabilities although we expect that the Corpora Camp platform will be able to accommodate both if parts of the platform go forward after Corpora Camp.

Processes can be thought of as information flowing from a source through a transformation and ending in a sink. For example, we can have the following:

Source	Discover and Source a document
Transformation	Analyze the document
Sink	Output the results of the analysis

Going the other direction:

Source	Form fields submitted in a browser
Transformation	Information vetted and filtered
Sink	Object Management (Creating, Updating, or Deleting



Most real-world processes will be a mix of these directions. For example, we may need to do some Discovery or Sourcing before we can do Object Management. At Corpora Camp, we expect the functionality to be composed primarily of the Modeling to Viewing flow.

Corpora Camp Platform

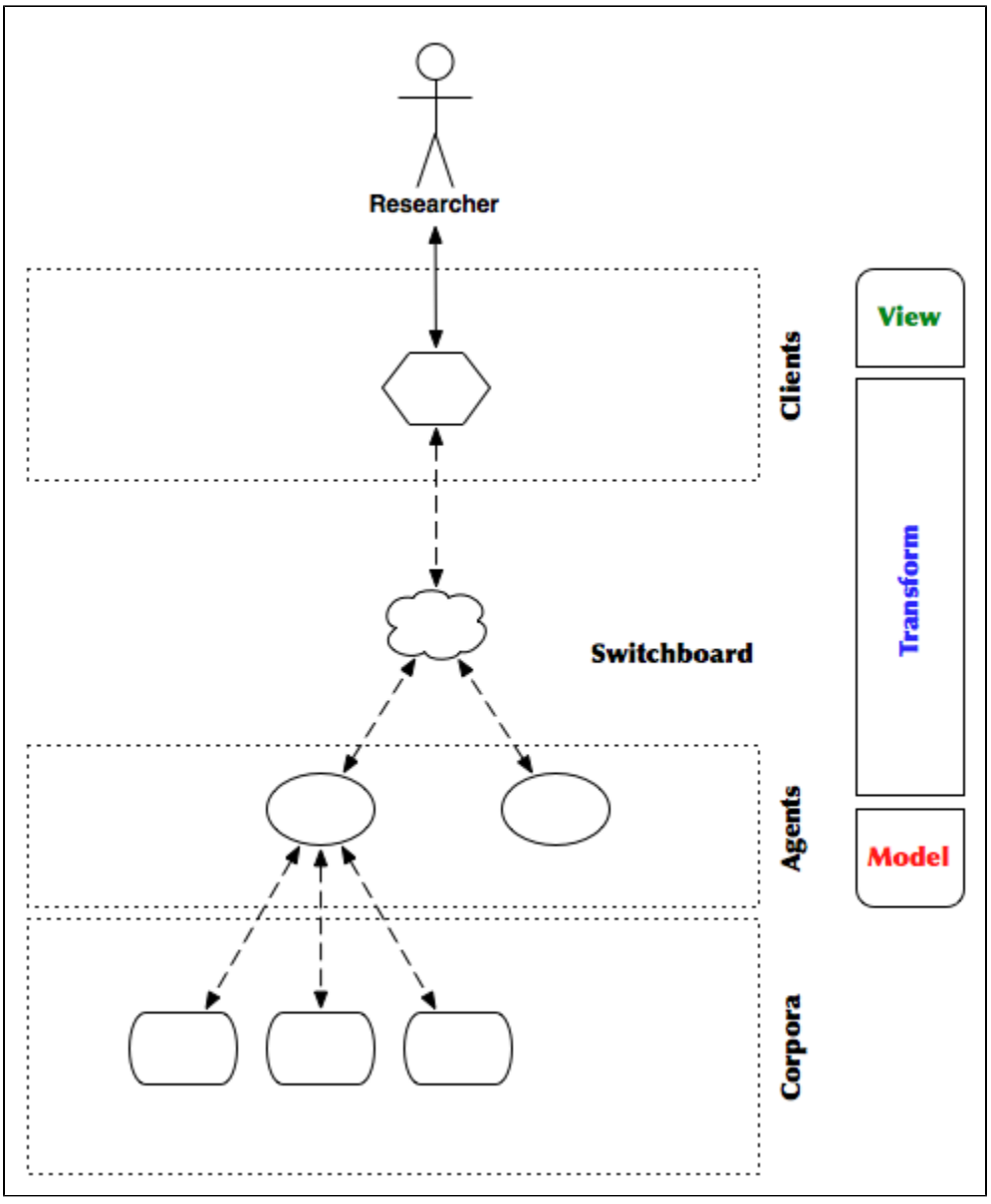
The platform for Corpora Camp is designed to imitate the type of distributed system envisioned by the larger Bamboo Project. The platform divides applications into three components: clients consuming information from the system (workspaces viewing and interacting), agents providing access to information (corpora interoperability managing the model), and agents providing access to transformations and analysis (services transforming). A switchboard server mediates the conversation among these components.

The primary principles behind this architecture are:

- Maximum agency for all participants: programmers, researchers, corpus curators, etc.
- Layered design enabling emergent behavior: each layer should build on the layer below and enable the layer above, providing the maximum flexibility into the future for developing new tools that will integrate into the existing infrastructure.

In the Corpora Camp platform, Clients act on the behalf of the Researcher, providing access to the ecosystem through the mediation of the Switchboard. The Switchboard acts as the go-between, broadcasting queries from the Clients to the Agents and returning responses from the Agents to the Clients. The Agents act as translators between various resources (Corpora) and the ecosystem.

Agents manage the various activities that fall under the Model rubric: Discovery, Object Management, and Sourcing /Citing. Moving these responsibilities to the Agent allows each Corpus to organize itself in the way that fits best the mission of the Corpus and associated research projects.

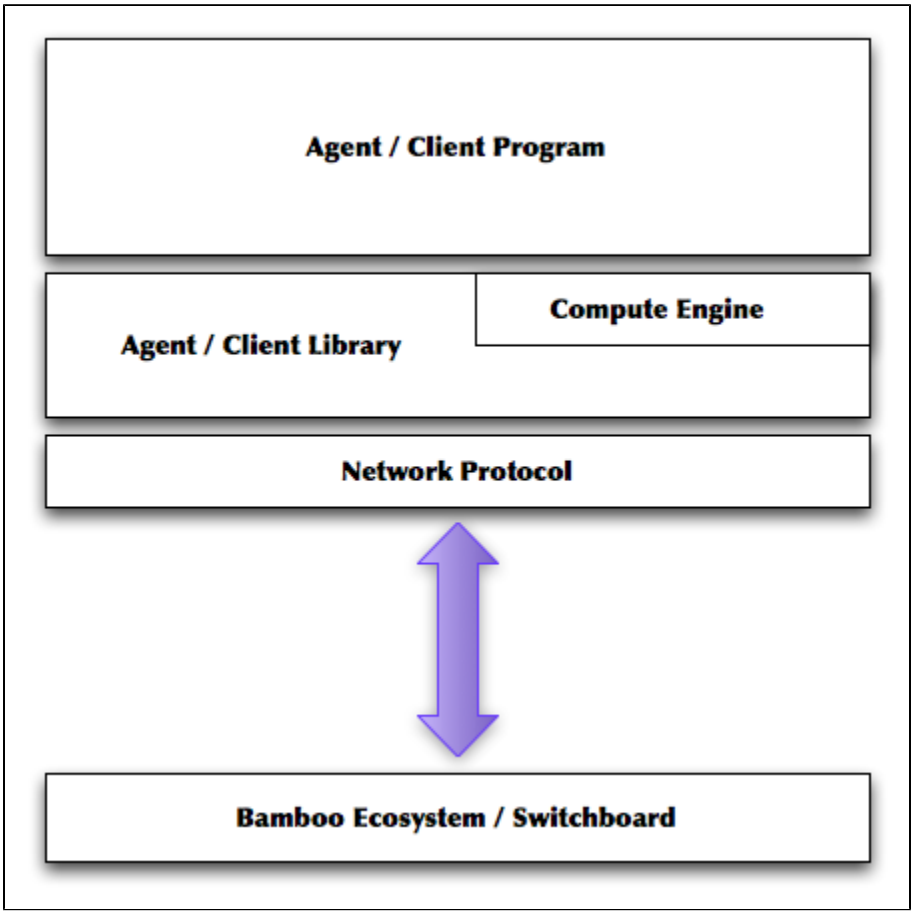


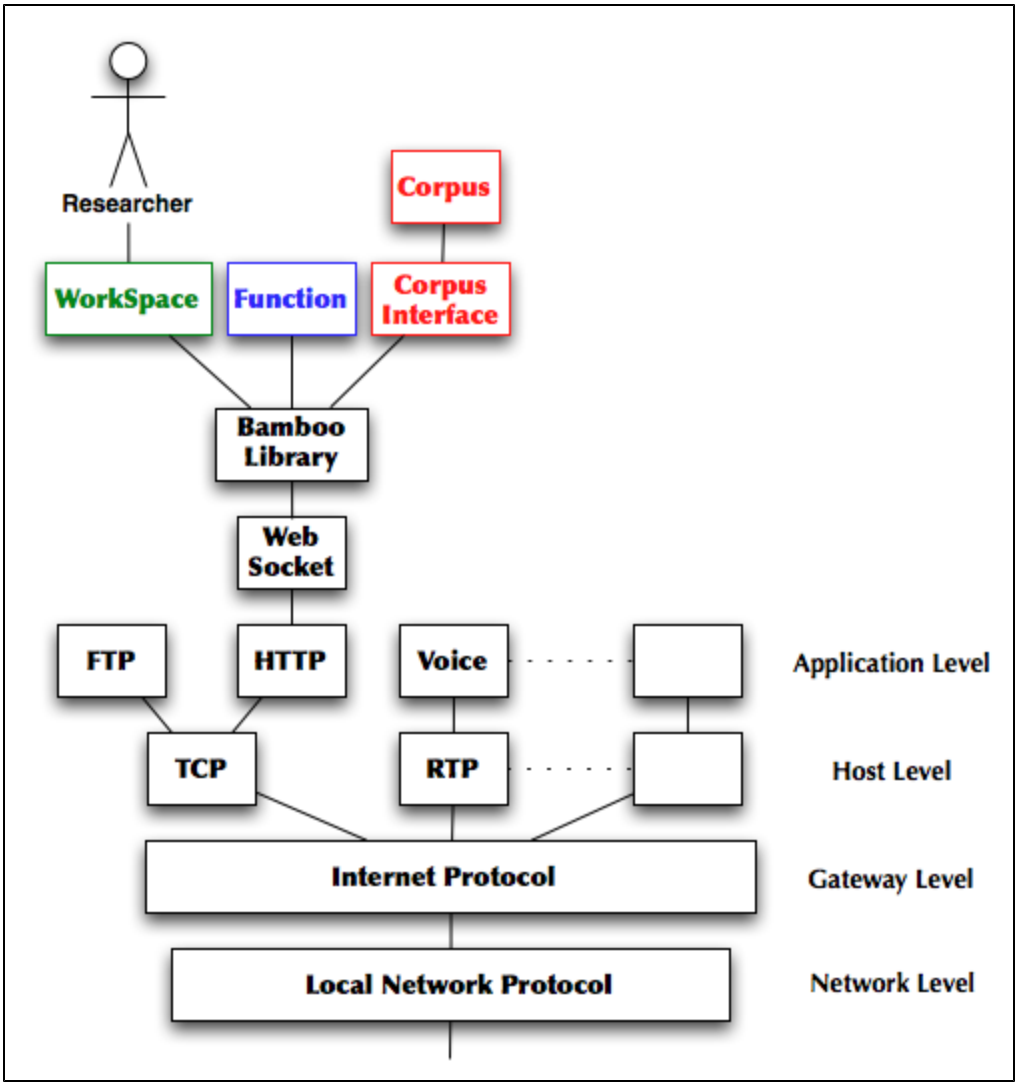
Platform Libraries

The core platform software consists of a set of libraries that implement the network protocol between Agents, Clients, and the Switchboard. A Compute Engine is included to ensure a common expression language across the Bamboo ecosystem. The Compute Engine is the basis for exposing functionality to other Clients and Agents, allowing the functionality needed in order to implement the Model activities.

By developing the platform as a series of layers, we can make changes in how any one layer is implemented without requiring comparable changes in other layers. For example, we could replace the WebSocket/HTTP/JSON layer with something like RabbitMQ.

We have initial development done in Perl for all platform components. We have a working client library in Ruby. A client library in JavaScript is in development.





Tag Libraries

The platform is designed to allow agents to provide arbitrary functionality to clients by offering collections of functionality to the platform. The platform informs clients of available functionality. The API libraries for clients and agents make this offering and acceptance of functionality relatively transparent. Once the libraries are completed, applications using a client to access the Corpora Camp platform will not need to distinguish between functionality provided by the application and functionality provided by an agent.

In general, libraries divide functionality up as follows:

	function	mapping	reduction	consolidation
Source	X	X		
Transformation	X	X	X	X
Sink				

Note that we don't have any Sinks for Corpora Camp. We are only expecting Source and Transformation functionality: search, retrieval, and analysis of corpora. The sink is the presentation in the client.

Example Tag Library

We'll use Ruby to illustrate how to build a tag library since the resulting library can be used with the JRuby environment as well.

N.B.: We are using the Utukku:: namespace so that we don't present the platform code for Corpora Camp as an official Bamboo Project product. Utukku is a name from Sumerian mythology.

```
class Example::TagLib < Utukku::Engine::TagLib
  namespace 'http://www.example.com/ns/taglib/1.0#'
```

A tag library is named by an XML namespace declaration. This allows us to name the interface definition we are expecting to use independent of the underlying implementation. As long as we reference the namespace, we can get to the functions in the library.

```
  mapping 'double' do |ctx, arg|
    arg.value * 2
  end
```

A mapping is a function that can be applied to all of its arguments simultaneously. The platform manages the simultaneity of execution for mappings. All the tag library has to do is define what the mapping does on a particular item.

```
  reduction 'count' do |ctx|
    { :init => proc { 0 },
      :next => proc { |count, v| count += 1}
    }
  end
```

A reduction is a function that takes a list of items and returns a value based on the complete list. For example, the count must be able to see each item to know how many items there are (arrays able to return their size are an optimization). The reduction provides procedures to the platform for initializing the reduction state and then modifying that state for each item passed to the reduction.

```
  consolidation 'count' do |ctx|
    { :init => proc { 0 },
      :next => proc { |sum, v| sum += v }
    }
  end
```

A consolidation is a function that takes the results of the reduction by the same name and returns a single result combining the results of the reductions. This allows us to break a reduction up into several parallel threads and then bring the results back together. In some map/reduce libraries, a reduction is expected to have the same properties as a consolidation, namely, the input data type and output data type are identical.

```
  function 'string-join' do |ctx, args|
    joiner = args[1].first.to(Utukku::Engine::NS::U, 'string').value
    Utukku::Engine::ReductionIterator.new(args[0], {
      :init => proc { [ ] },
      :next => proc { |acc, s|
        acc.push(a.to(Utukku::Engine::NS::U, 'string').value)
      },
      :finish => proc { |acc| acc.join(joiner) }
    })
  end
```

A function can be used when the functionality isn't a mapping, reduction, or consolidation. It's more general purpose than the others, but the platform does less work for a function.

```
end
```

That's all there is to providing the functionality shown. Note that the library doesn't need to worry about how it's being used: either synchronously or asynchronously, local or remote. The platform library handles all of those things internally based on the context in which it is being used.

If the library was loaded and the XML prefix ‘x’ was assigned to the namespace ‘http://www.example.com/ns/taglib/1.0#’, then the expression ‘x:double(1 .. 10)’ would return the list of integers ‘2, 4, 6, 8, 10, 12, 14, 16, 18, 20’. The expression ‘x:count(1 .. 10)’ would return ‘10’. The expression ‘x:count*(x:count(1..10)|x:count(2..20)|x:count(3..30))’ would return ‘57’. If the library was exported by an agent, then the same expression in a client would produce the same results, even if it required a few trips across a network.

Accessing a Tag Library Remotely

If an agent has exported a tag library, then the switchboard server will notify the client of the namespace and available functions when the client connects. The typical client will create some stub objects to handle the transition to the network when using remote functionality. Not all client libraries will be able to do this for Corpora Camp. This section will explain how to interact with a remote function ‘by hand.’

The underlying protocol is JSON sent over WebSocket. This allows JavaScript clients in browsers that support the emerging HTML 5 standards (e.g., Chrome) to connect directly to the platform.

Each packet consists of three keys: ‘class’, ‘data’, and ‘id’. The ‘class’ is mandatory. Without it, the client, agent, and switchboard wouldn’t know what to do with the packet. The ‘data’ is dependent on the ‘class’. The ‘id’ is mandatory for any ‘class’ that expects a response or works with a conversation thread. The ‘id’ is auto-generated by the libraries.

Remote function calls are represented as flows in the platform. The typical flow cycle begins with the client creating the flow and then providing it data. Agents may respond at any point depending on the nature of the function being executed by the flow. The client signals that it has provided all data for the flow and then waits for the agents to finish producing their results. Once data has been provided and results have been produced, the client can close the flow, signaling that any resources in the switchboard and agents can be freed. The client also can close a flow at any time, discarding any pending results from agents.

The following Ruby code will bypass the Compute Engine and call the ‘x:double’ function directly (regardless of where the functionality resides):

```
Utukku::Client.new('ws://localhost:3000/demo') do |client|
  ns = 'http://www.example.com/ns/taglib/1.0#'
  client.function(ns, 'double', [
    Utukku::Engine::ConstantRangeIterator.new(1, 10)
  ], {
    :next => proc { |v| puts "  doubled to #{v}" },
    :done => proc {     puts "all done with the doubling" }
  })
end
```

Using the compute engine, the code could look like this:

```
Utukku::Client.new('ws://localhost:3000/demo') do |client|
  context = Utukku::Engine::Context.new
  context.set_ns('x', 'http://www.example.com/ns/taglib/1.0#')

  parser = Utukku::Engine::Parser.new

  expr = parser.parse('x:double(1..10)', context)
  expr.async(context, false, {
    :next => proc { |v| puts "  doubled to #{v}" },
    :done => proc {     puts "all done with the doubling" }
  })
end
```

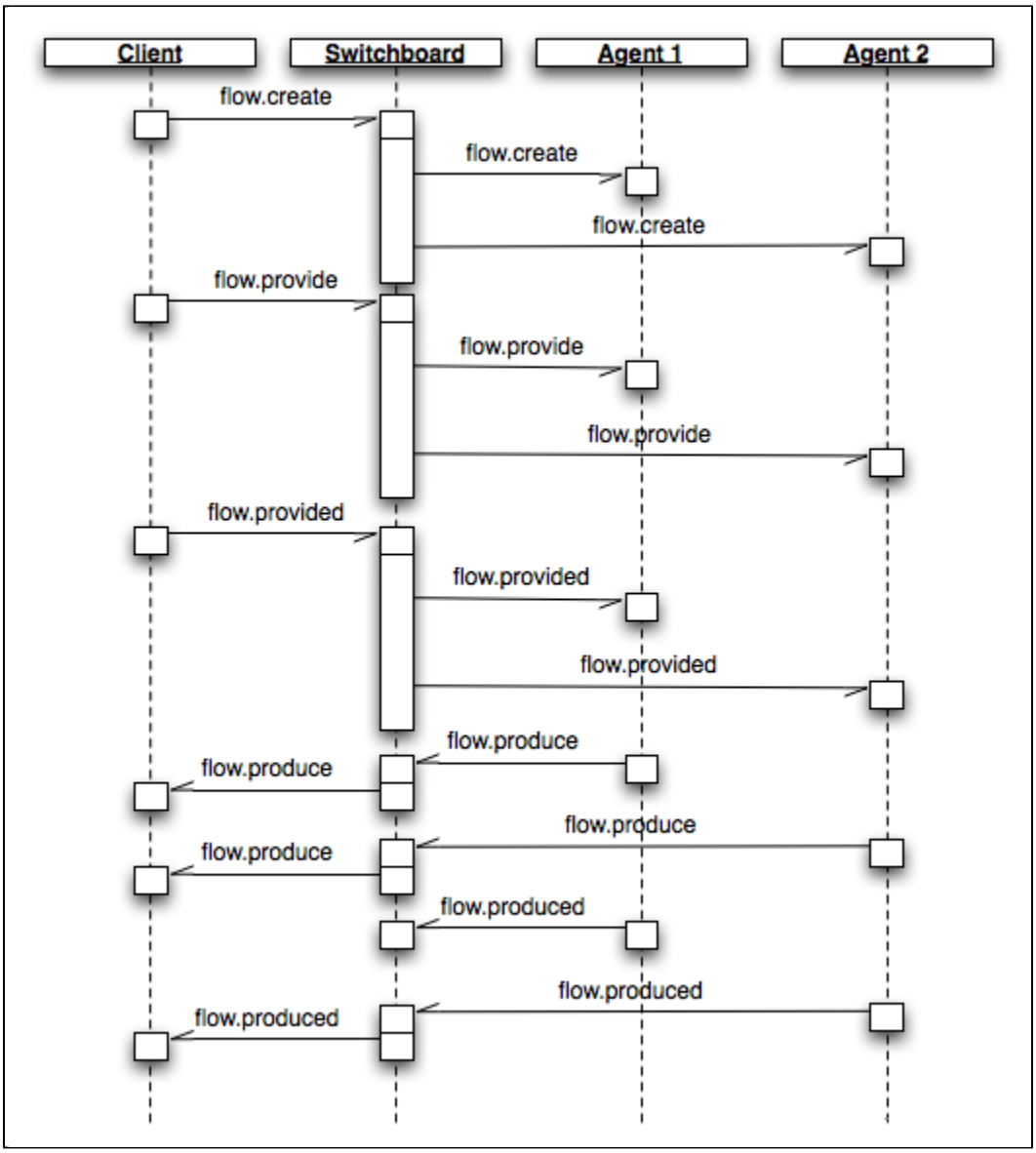
A similar pattern holds for other languages as well. For example, in JavaScript, we might have:

```
Utukku.Client.Connection({
  url: 'ws://localhost:3000/demo',
  onSuccess: function(client) {
    var ns = 'http://www.example.com/ns/taglib/1.0#',
        handler = Utukku.Engine.TagLib(ns),
        iterator = handler.function_to_iterator('double', [
          Utukku.Engine.ConstantRangeIterator(1, 10)
        ]);
    (iterator.async({
      next: function(v) { /* do something with v */ },
      done: function( ) { /* finish up */ }
    }))();
  }
});
```

We could fold all of this into a simple interface:

```
Utukku.Client.Function({
  url: `ws://localhost:3000/demo',
  namespace: 'http://www.example.com/ns/taglib/1.0#',
  name: 'double',
  iterators: [ Utukku.Engine.ConstantRangeIterator(1,10) ],
  next: function(v) { /* do something with v */ },
  done: function( ) { /* finish up */ }
});
```

The JavaScript library can cache the client connection and reuse it for any requests that have the same WebSocket URL.



Exporting a Tag Library

A Perl agent can export a tag library very simply:

```
my $agent = Utukku::Agent -> new(
  url => 'http://localhost:3000/demo',
  namespaces => ['http://www.example.com/ns/taglib/1.0#'],
);

$agent -> run;
```

The Perl version needs a ‘http’ URL instead of a ‘ws’ URL due to the Perl libraries we’re using. This is a minor bug.

The Ruby version could be:

```
Utukku::Agent.new do
  url 'ws://localhost:3000/demo'
  export_namespace 'http://www.example.com/ns/taglib/1.0#'
end
```

This right-side column is an artifact of "old-style" navigation. **Please remove sections, columns, and content of the right-side column when this page is next edited.** Cf. the Wiki navigation changes - July 2011 page for a 'how-to' on removing old-style page markup.