IoTPy: A Versatile Framework for Streaming Applications

Rahul Bachal

Table of Contents

1. Introduction

Real-time processing of data has become critical for decision making in many areas as the number of internet connected sensors embedded in a variety of devices has increased by many orders of magnitude. Sensors produce continuous streams of measurements that need to be analyzed in real-time and decisions need to be made on the basis of such analysis. For example, algorithmic trading used by many brokers and hedge funds must analyze stock ticker data within microseconds to inform the next trade. Similarly, determining the center of an earthquake rapidly, or even better, predicting shaking from an earthquake even a few seconds in advance, can save millions of lives. However, doing so requires analyzing vibration information sent from multiple geographically distributed sensors in real-time.

The number of sensors and devices connected to the internet has grown exponentially in recent years, leading to the formation of what is known as the Internet of Things (IoT). The IoT is characterized primarily by the ubiquity of devices that, while being interconnected, are also smart [5]. The data sent by such sensors is primarily comprised of continuous lists of values that are time-based, or streams of data [9]. Due to the large number of sensors being added to the IoT and the amounts of data being streamed by such sensors, the total data that needs to be processed continues to grow rapidly. The area of computer science that deals with such large data sizes and scale is referred to as Big Data [4]. The sensor data streams are used to analyze and make decisions in real-time. Streaming data can be analyzed using statistical methods or using machine learning.

Machine learning (ML) attempts to learn the characteristics of a system or process for cases where it is known that there is pattern between the inputs and outputs of a system but where the mathematical relationship between the inputs and outputs is not known and cannot be extracted [6]. Supervised ML uses known input-output data (or training data) to *learn* a model of the real-life process. Once such a model is trained, new unknown inputs can be fed to the model and the model outputs predicted results for these new inputs. For example, predicting the center of an earthquake based on shake data from sensors requires using data from past earthquakes to train the model with shake data and location of sensors as input and center of earthquake as the output. Once the model is trained, it can be applied to new shake data from sensors to predict the location of an earthquake that has just started [10]. Model training can be done in batch mode by collecting and using past data. However, due to the rapid pace of data streams and the rapidly changing nature of the system or process being modeled, such a model quickly becomes obsolete. Hence, there is a need to for online ML which requires continuous retraining of the model using incoming streams of data while simultaneously predicting outputs from the model. Figure 1 shows a simple outline of machine learning being applied to 3 input streams, processing the data in real time, and resulting in an output stream which is shown as the prediction stream.

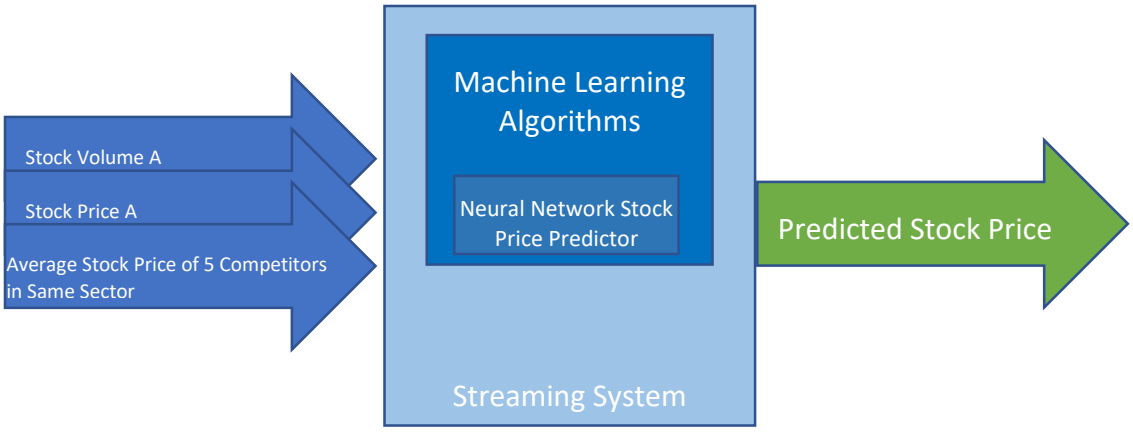

Figure 1: Machine learning applied over streaming data to predict stock price for company A. The arrows on the left represent three input streams with different data. The first stream is the volume of shares traded per minute. The second stream is the average price of the stock over a minute. The third stream is the average stock price of 5 competitors in the same sector over the same time period. This is passed to a streaming system that applies machine learning algorithms to the streams and produces an output stream that is a prediction of the stock price. A neural network is a type of machine learning algorithm that is used for this application.

Many machine learning algorithms exist and implementations are readily available, such as *scikit-learn* [12] and *Pandas* [11]. However, these algorithms are designed to work on static data. For example, an algorithm may take a list of values as input but it requires all values in the list to be available before the model can be evaluated. Applying such algorithms for streaming data is not straightforward and requires complex programming. Thus, this confluence of ML and streaming data presents a unique challenge as it requires users to understand two complex technologies – machine learning and streaming systems.

Common software programming languages and frameworks are focused mainly on batch data processing, where programmers write functions and code to handle static, offline data. To develop applications that can analyze data in real time, a streaming system is necessary. Such a system needs to at a minimum, handle data streams as first-class objects, provide memory management, and provide an interface for programmers to interact with data streams. This is commonly referred to as stream processing. There are a few systems and frameworks that are designed specifically for streaming applications and big data. Most of these systems operate exclusively in the cloud; all computation is done by collections of computers in a data center. Efficient processing of data streams requires distributed computing, with some computation done by smart sensors, more computation done at distributed aggregation points, and final computation in the cloud.

Apache Spark [2] allows users to run parallel applications on big data sets and enables stream processing with Spark Streaming, but a drawback is the steep learning curve for users to learn Apache Spark's custom API and the difficulty of using the system across widely distributed systems. Similarly, Apache Storm [3] is a distributed system that allows computation to be distributed across computers, but the setup overhead is significant and makes it difficult for novice users. Apache Hadoop [1] is a MapReduce system for processing big data sets but is mainly designed for batch processing and not streaming. Millwheel [8] allows users to build

computational graphs in distributed systems to process streams, but is designed for advanced users with expertise in distributed systems and does not provide a simple way for novice users to process streams. All these examples show that existing stream processing systems are built for the expert user and have steep learning curves. In addition, none of the systems offer first class support for integrating online learning of ML models as well as evaluation of trained model for prediction on streaming data inputs.

We present a system called IoTPy. This system allows stream processing for non-programmers, novice Python programmers, and experts in streaming systems. Non-programmers can write streaming applications by using modules and templates written by other users. Novice Python programmers can write Python code for batch data that is converted to streaming code. Streaming experts can write algorithms designed for streaming data. Our system enables users to write and use streaming applications, including ML, for data in real time.

The work done in this thesis focuses on three components of IoTPy. First, we describe a machine learning framework that is layered on top of the core abstractions of IoTPy to enable novice programmers to apply ML algorithms written for static data to streaming application. Second, we describe a declarative way to define streaming application templates, a language for defining complex streaming applications allowing reuse of granular stream processing parts or templates and a parser that deserializes the language definition into a runtime format. Finally, we describe an assembler that puts together the parts defined using the declarative language and a runtime for executing templates in a single process, across multiple processes on a single machine and in a distributed computing environment across many machines.

2. Machine Learning

2.1 ML Framework

Machine learning (ML) attempts to learn the characteristics of a system or process for cases where it is known that there is pattern between the inputs and outputs of a system but where the mathematical relationship between the inputs and outputs is not known and cannot be extracted [6]. Such a pattern is learned by training machine learning models, which are mathematical functions in a defined hypothesis space that are used to predict output values for input values.

A machine learning algorithm has two components: training and predicting. Given a training dataset, the algorithm first trains a model. It then uses the model to generate predictions. For example, predicting the center of an earthquake based on shake data from sensors requires using data from past earthquakes to train the model with shake data and location of sensors as input and center of earthquake as the output. Once the model is trained, it can be applied to new shake data from sensors to predict the location of an earthquake that has just started [10].

There are two main classes of machine learning: supervised learning and unsupervised learning. Supervised learning refers to scenarios where the training dataset has labeled outputs and the goal is to learn these outputs generally. The previous example using earthquake data uses supervised learning. Unsupervised learning refers to scenarios where the training dataset does not have labeled outputs and the goal is to learn patterns in the data. For example, in credit card fraud detection, the training data does not have any labels regarding if a transaction was an anomaly or not. Instead, the machine learning algorithm attempts to learn the underlying patterns in the transaction history to determine if the current transaction is an anomaly.

To abstract this process, we split machine learning algorithms into two components: training and prediction. This framework allows us to plug-and-play any machine learning algorithm by accepting user-defined functions that run training and prediction respectively. To use the framework effectively, you to take the following steps:

1. Determine the type of function desired to learn.
2. Implement a training function that uses a training dataset to train the function.
3. Implement a prediction function that uses the trained hypothesis function to generate a prediction value for an input.

These steps are general to machine learning; it is important that you recognize and separate a machine learning algorithm into these specific components.

The ML framework is extremely flexible, supporting both supervised and unsupervised learning. You only need to ensure that the training and prediction functions provided are consistent with each other.

There are two main ways you can use this framework. You can use functions defined in machine learning modules without any significant coding or you can write your own machine learning functions. We focus first on the simpler of these two – using functions defined in other machine learning modules.

There are many Python modules for machine learning. For example, *scikit-learn* [12] offers various functions and algorithms for supervised and unsupervised learning. To use these functions, you must write training and prediction functions that wrap them.

2.1.1 Structure of a Training Function

A training function has the following structure:

```
def train_function(x, y, model, window_state):
    if not model:
        # Initialize model - constructor may be
different
        model = Model()

    # Computations

    # Return model
    return model
```

x and y are numpy arrays containing the training values. model is the model returned by this function. window_state describes the training window and is not needed. All training functions will have the same signature. This enables the function to define and maintain an internal state to save the machine learning model. We first check whether the provided state is already initialized. If it is not, then we initialize it by defining a class and setting it to be an instance of this class. This allows the state to hold any variables necessary for the machine learning model. At the end of this function, we return the updated state. This will be passed to the training function for the next window, allowing us to maintain the state between window transitions.

2.1.2 Structure of a Prediction Function

A prediction function has the following structure:

```
def predict_function(x, y, model):
    # Computations

    # Return prediction value
    return value
```

All prediction functions will have the same signature. The prediction function receives the current state containing the machine learning model as updated by the training function. We can use this to predict a value for a given input. This function returns the prediction value. This function should not modify the state – the only function that modifies the state is the training function.

We begin with a simple example of supervised learning.

Linear regression is a machine learning algorithm that tries to learn a linear model. Figure 2 shows an example of linear regression.

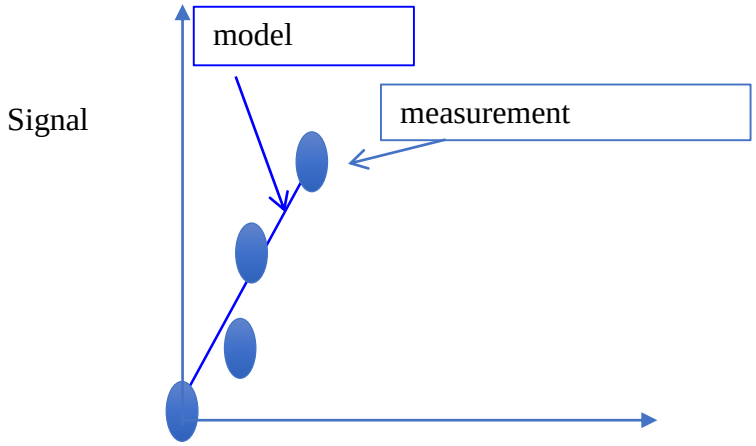

Figure 2: Example of linear regression. The dots represent measurements and the line represents a trained model.

To run linear regression from *scikit-learn* [12], you must first write a training function. This function looks like:

```
def train(x, y, model, window_state):
    regr = linear_model.LinearRegression()
    regr.fit(x, y)
    return regr
```

This function uses *scikit-learn* to create a linear model trained on data and returns it.

Next, you must write a prediction function. This function looks like:

```
def predict(x, y, model):
    return model.predict(x)
```

The parameter model refers to the model returned from the training function.

You can use any function from *scikit-learn* or other modules in a similar fashion.

For example, we can run linear regression as follows:

```
linear_regression(x:stream, y:stream) -> z:stream {
    model = train(in=x, function=train,
num_features=2, min_window_size=5, max_window_size=30,
step_size=2);
    z = predict(in=y, function=predict, model=model,
num_features=2);
}
```

We discuss how to write this template later.

2.2. Incremental Algorithms

The ML framework allows users to run any machine learning algorithm by writing their own training and prediction functions or using existing algorithms such as *scikit-learn*. We describe two machine learning algorithms that we have developed from scratch for streaming data. These algorithms have similar functionality as their standard counterparts written for static data. However, we show that our algorithms are more efficient and show better prediction accuracy. These functions also illustrate the extensibility available in IoTPy for expert developers who have an understanding of streaming systems.

The design of incremental algorithms can be explained using a simple example. When a sliding window moves over a stream, most of the values in the window stay the same. Consider a sliding window that looks at a stream $[1, 5, 3, 6, 4, 2 \ldots]$. A window with size 5 and step size 1 will look like $[1, 5, 3, 6, 4]$, then $[5, 3, 6, 4, 2]$, and so on. From the first window to the next, we lose one value from the beginning, 1, and add one value to the end, 2. The rest of the values in the window stay the same. We can reuse computation between windows to learn more efficiently. This leads us to create native streaming machine learning algorithms that learn incrementally. These provide better performance and in some cases improved prediction accuracy due to the incremental learning.

Two such algorithms were developed: linear regression and kmeans. These algorithms were chosen for native streaming implementation as these are widely used for machine learning applications.

2.2.1 Linear Regression

Linear regression is a ML algorithm that attempts to fit a linear function to model data. Ordinary least squares linear regression is solved by the equation $w = (X^T X)^{-1} X^T y$, where X is the data matrix with dimensions $n * m$, y is the output vector with dimensions $n * 1$, and w is the weight vector with dimensions $m * 1$ for n = number of data points and m = number of features. This approach works for 1 feature and hence for data with 1 feature, we do linear regression using incremental matrix inversion. For data with more than 1 feature, however the cost of matrix inversion makes least squares linear regression unusable. We chose stochastic gradient descent algorithm for data with more than 1 feature. Stochastic gradient descent uses the gradient of the error function to find the optimal parameters of the model until a local minimum is found. Our implementation assumes that the local minimum for the weight vector does not change significantly when the sliding window shifts. This assumption is valid as long as the underlying data comes from similar distributions.

2.2.2 KMeans

The KMeans algorithm works by initializing k centers randomly and then assigning each point x to its closest center. Each center is then updated to the mean of the data assigned to it. The algorithm iterates until the centroid assignment stabilizes. Mathematically, the algorithm performs the following steps:

- Initialize k random clusters $m_1, \ldots, m_k$.

- Partition the data into k sets $S_1, \ldots, S_k$ where for each S_i, $S_i = \{x_j : \|x_j - m_i\|^2 \leq \|x_j - m_l\|^2 \forall l, 0 \leq l < k$
- Update each cluster $m_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j$

The iteration repeats Steps 2 and 3 until the centroid assignment stabilizes. The vanilla batch algorithm performs the above steps for each shift of the window. However, in streaming data, since the data distribution is the same, the centroids do not move very far between windows. This allows us to improve the efficiency of the algorithm by using the previous centroids as the input for the next window. This ensures that if the new centroids are close to the previous ones, then the number of iterations needed to converge will be minimal.

2.3 Examples

We provide three examples of ML. To run these, type:

```
IoTPy assemble linear_regression
IoTPy assemble meetup
IoTPy assemble twitter
```

The meetup template runs kmeans on RSVPs from Meetup.com and plots the centroids on a map of US. The twitter example runs sentiment analysis on tweets from Twitter [16] and plots them on a map of US.

Next we describe the declarative way to define streaming application templates, a language for defining complex streaming applications allowing reuse of granular stream processing parts or templates and a parser that deserializes the language definition into a runtime format.

3. Declarative Streaming Applications

Enabling non-programmers to create streaming applications requires the creation of a declarative format for stitching together stream processing parts into a streaming application without needing any code. We describe below a simple declarative format for defining streaming application templates with reusable parts.

3.1 Template Definition

A template is a collection of parts with inputs and outputs. Each part is defined as its own template and has its own inputs and outputs. For example, we may define a template called **test** with input *x*. This input is passed to **map** with the function *times_two* to multiply each value in *x* by two. Finally, the output of **map** is returned as stream *y*. This looks like the following:

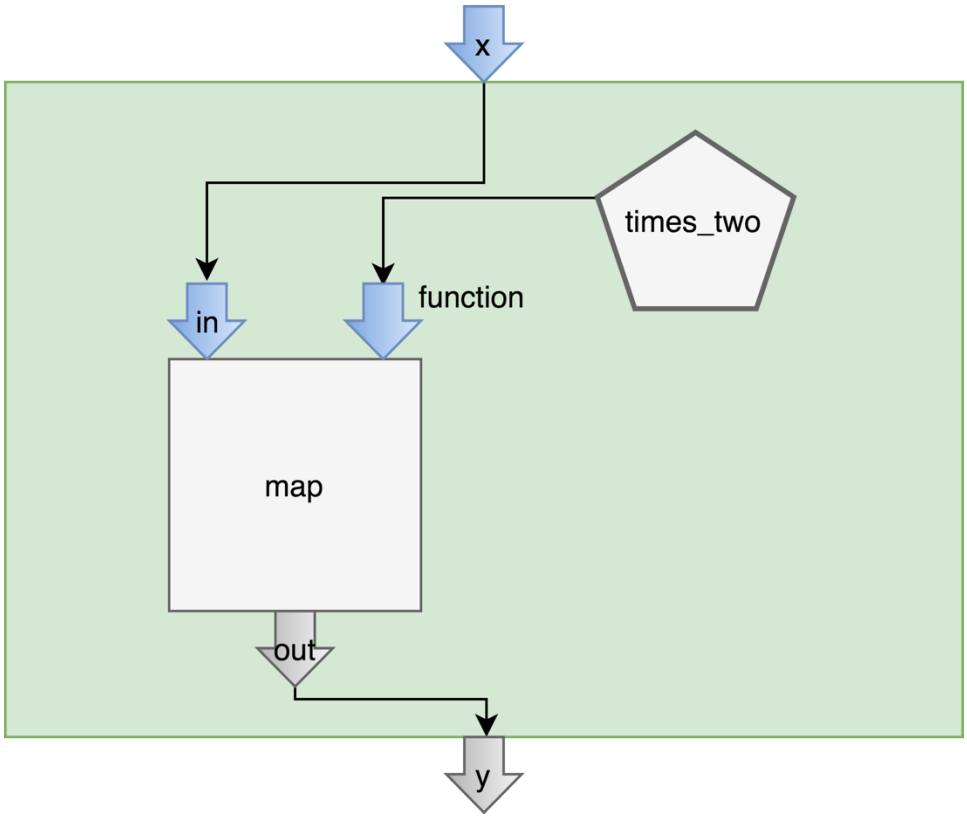

Figure 3: Example of a template. This template takes an input stream, multiplies the values by 2, and returns the results.

Templates are defined as a structured data with a fixed schema and are serialized to JSON [14]. The schema has the following keys:

- name
  - The name is the name of the template
- assembly
  - assembly specifies whether this template is a collection of subparts or if it is a leaf template. For example, the template test will have a value of assemble for assembly. The template for map will have a value of map for assembly.

- Inputs
    - The inputs is a list of input parameters for the template. Each parameter is either a string describing its name or a list [type, name], where type is a string specifying the type and name is a string specifying the name.
- outputs
    - outputs is specified in the same manner as inputs.
- components
    - components is a list of subparts in the template, where each value in the list is a list describing the name of the part and the type.
- optional
    - optional is a list of parameter names that are optional.
- externals
- internals

These keys specify the parts of the template. *externals* and *internals* is used to specify the values for each subpart. We describe each of these below.

3.1.1 externals

The *externals* key specifies how external parameters connect to subparts in the template. This key contains a list of values, where each value specifies a connection. Each value is specified in the format:

```
[external name, part name, parameter name]
```

For example, in the template test, we pass input *x* and the function *times_two* to **map**. We specify that *y* is the same as **map** out. This looks like:

```
[x, map, in],
[“times_two”, map, function],
[y, map, out]
```

externals can be used to specify constant values for parameters, as shown in the previous example. In this case, the name of the function *times_two* is a constant and is specified with quotation marks. Integers and floats can also be specified as constants.

So far, we have shown that the *externals* key allows us to specify 1:1 connections. However, some templates such as **zip** take in multiple parameters for the same input. For **zip**, multiple streams are sent to the input called *in*, as shown in Figure 4.

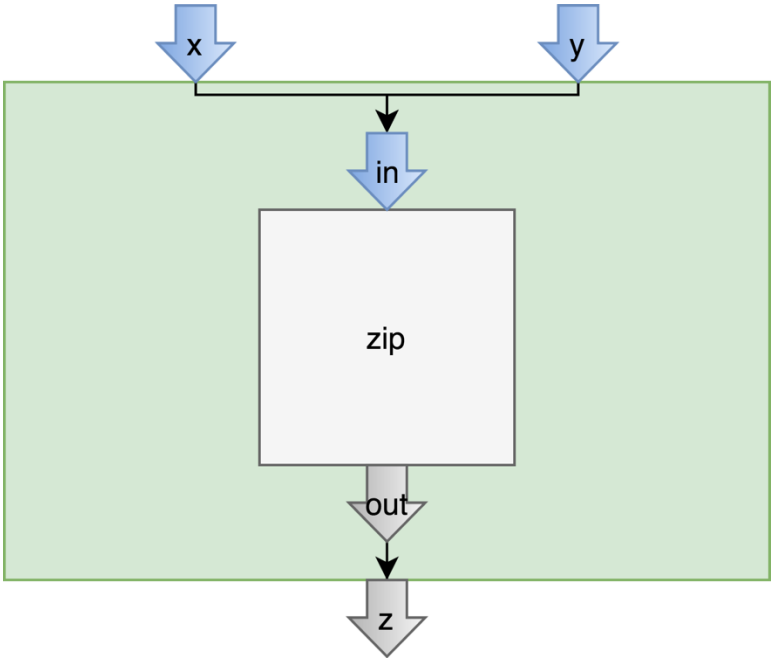

Figure 4: Example of many to 1 connection. This template takes streams x and y, puts the i^{th} values of each stream together, and returns the results.

We can specify this with *externals* by adding an optional index. For the previous example, this looks like:

```
[x, zip, in, 0],
[y, zip, in, 1],
[z, zip, out]
```

This specifies that the parameter *x* goes to the 0^{th} index for **zip** *in*, and parameter *y* goes to the 1^{st} index for **zip** *in*. We also specify that parameter *z* is the same as **zip** *out*.

Adding this index lets us specify many to 1 connections.

3.1.2 internals

The *internals* key specifies how parts are connected internally. This key contains a list of values, where each value specifies a connection. Each value looks like:

```
[source part name, source parameter name, destination
part name, destination parameter name]
```

This is the same general format as externals but only specifies connections unlike *externals*. *externals* can be used to specify constants. For example, consider the template shown in Figure 5.

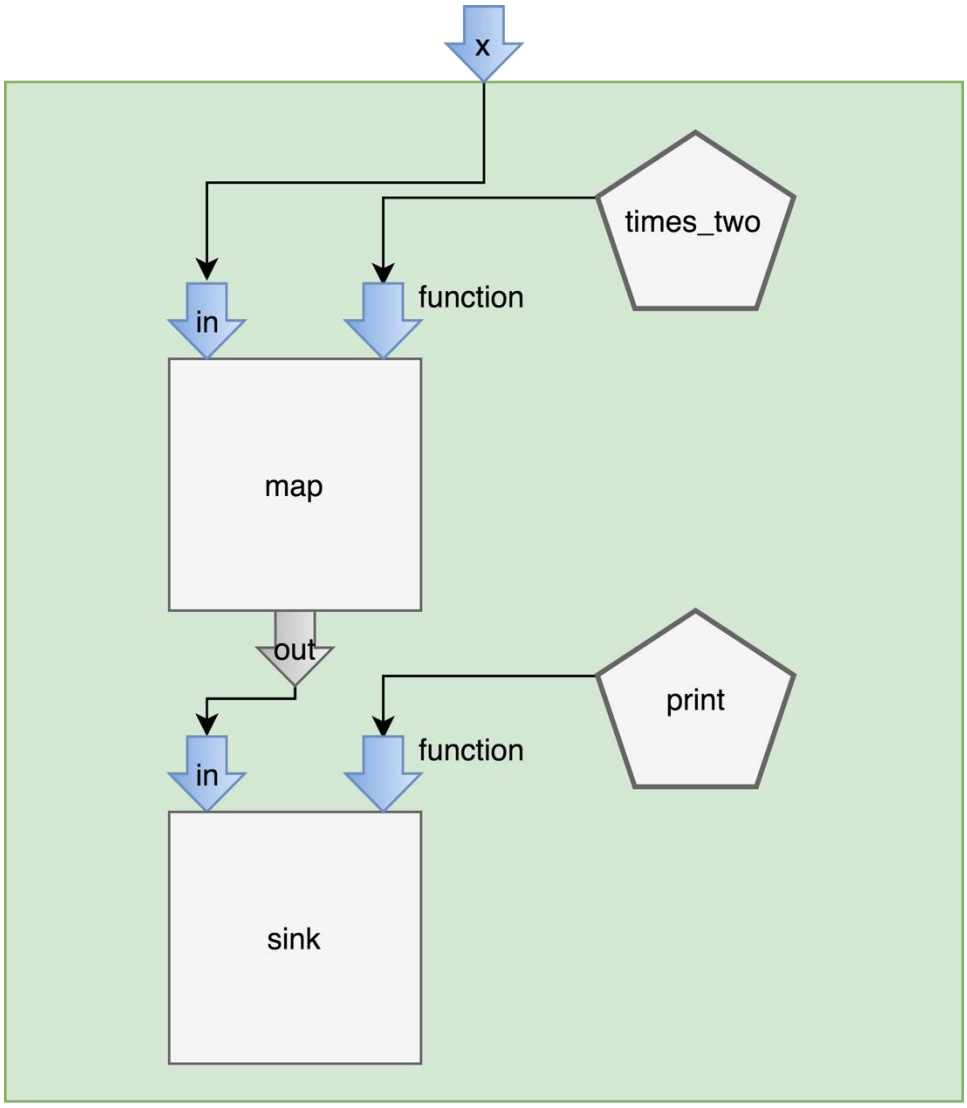

Figure 5: Example of template with subparts. This template takes a stream x, multiplies each value in x by 2, and prints them.

The *internals* for this looks like:

```
[map, out, sink, in]
```

This specifies that the parameter called *out* from **map** goes to the parameter called *in* for **sink**.

So far, we have shown that *internals* allows us to specify 1:1 connections. Like *externals*, we can also specify many to 1 connections. For example, consider the template shown in Figure 6.

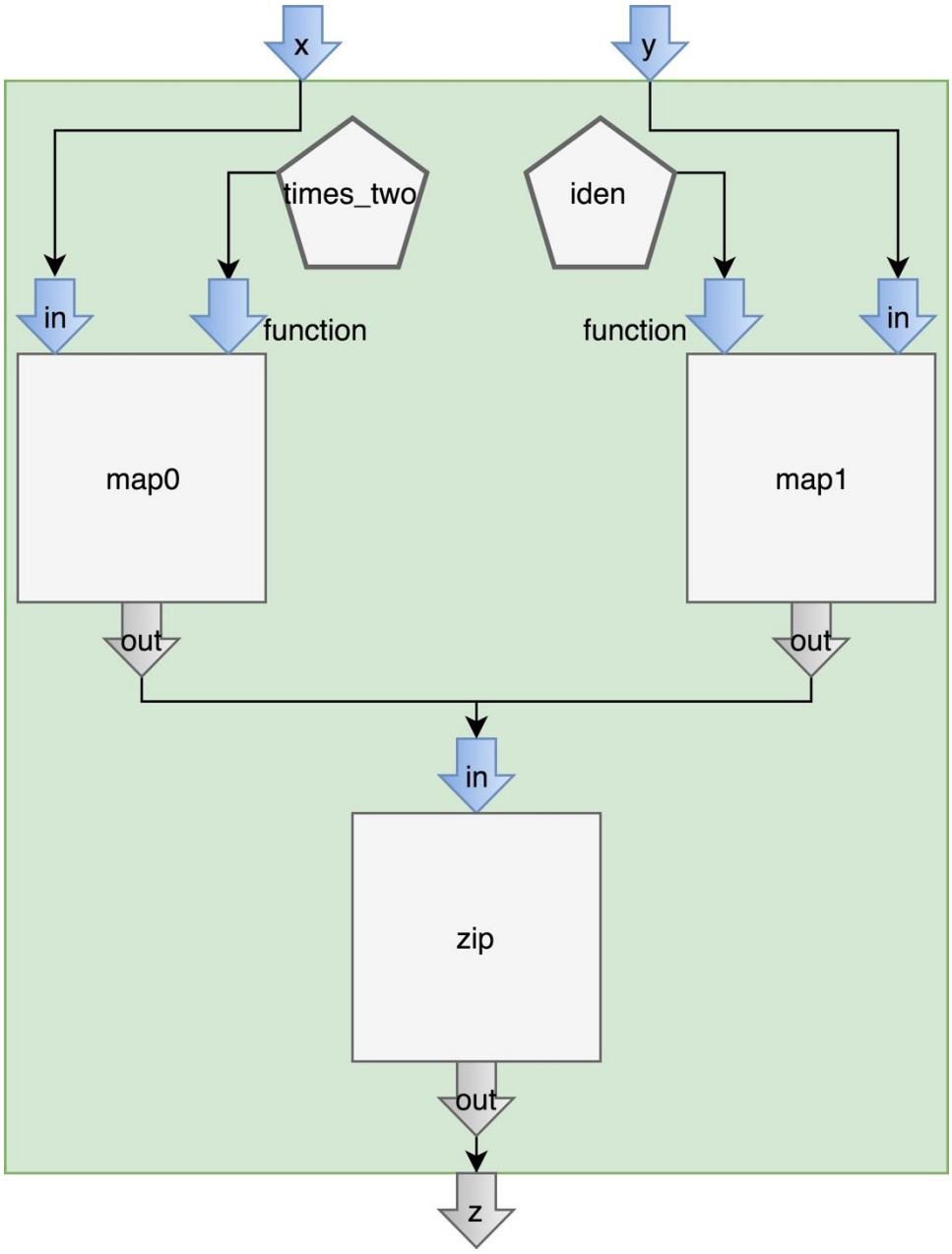

Figure 6: Example of internals many to 1 connection. This template takes streams x and y, multiplies values in x by 2, combines them with y, and returns the results.

The internals for this looks like:

```
[map0, out, zip, in, 0]
[map1, out, zip, in, 1]
```

By adding an index, we specify that the connection goes to an index in a list. An index can also be used for the source part.

We present the full JSON for the template:

```
{u'assembly': u'assemble',
 u'components': [[u'map0', u'map'], [u'map1', u'map'],
[u'zip0', u'zip']],
 u'externals': [[u'z', u'zip0', u'out'],
                [u'x', u'map0', u'in'],
                [u'"times_two"', u'map0',
u'function'],
                [u'x', u'map1', u'in'],
                [u'"iden"', u'map1', u'function']],
 u'inputs': [[u'stream', u'x'], [u'stream', u'y']],
 u'internals': [[u'map0', u'out', u'zip0', u'in', 0],
                [u'map1', u'out', u'zip0', u'in', 1]],
 u'name': u'test',
 u'optional': [],
 u'outputs': [[u'stream', u'z']]}
```

3.2 Parser

The previous section described a simple, declarative JSON format to define templates. However, as templates get more complex, the JSON becomes more complicated and difficult to read. This can be time consuming to create and error prone. Hence, we have defined a language in order to generate the JSON from a simple specification. This section describes the parser and syntax of this language.

A template written using this language has a name, inputs, outputs, and a list of lines describing the internal parts. This looks like:

```
name(arguments) -> outputs {
lines
}
```

The arguments for the template are specified as a comma separated list of names. We can also specify a type for an argument by writing it as name:type. The list of outputs is the same format. We can also specify optional parameters by adding an asterisk after the name. If there are no output parameters, we specify *outputs* as *None*.

Lines is a list of lines delimited by a semicolon that specify how the internal parts work. Each line is represented as a function call. The format is:

```
outputs = template(inputs);
```

where *outputs* is a list of outputs and *inputs* is a list of inputs for a template called template. Each input in *inputs* can be a name or can be a list of names.

We can take our previous JSON example and write it using this language:

```
test(x:stream, y:stream) -> z:stream {
    a = map(in=x, function="times_two");
    b = map(in=x, function="iden");
    z = zip(in=[a,b]);
}
```

To parse this definition, we use a Python package called pyparsing [17]. pyparsing is a recursive descent parser. We find tokens and create parsers for those tokens, and then combine these parsers. For example, the parser for the first line looks like:

```
first = name + inputs + arrow + outputs
```

This parser parses the JSON to a dictionary. To serialize the in-memory definition to JSON, we parse this dictionary in the following way:

For each line, check the outputs

If output is in template outputs, add it to externals

Else, add it to a dictionary as a temporary variable

For each line, check the inputs

If input is in template inputs, add it to externals

Else, find input in dictionary and add it to internals

We check if any outputs are unused and add an external connection that goes to None.

There are other specification languages that can be used to define event rules, such as TESLA [21]. TESLA defines rules in terms of events and notifications. Our language is simple and powerful as we define templates in terms of Python code and provide a full runtime to execute these templates in a variety of distributed and non-distributed configurations.

4. Assembling parts

As we have discussed, a template is comprised of other templates or encapsulation functions. Each template is stored as JSON in a TinyDB [18] database. The template structure leads to a recursive function for assembling parts.

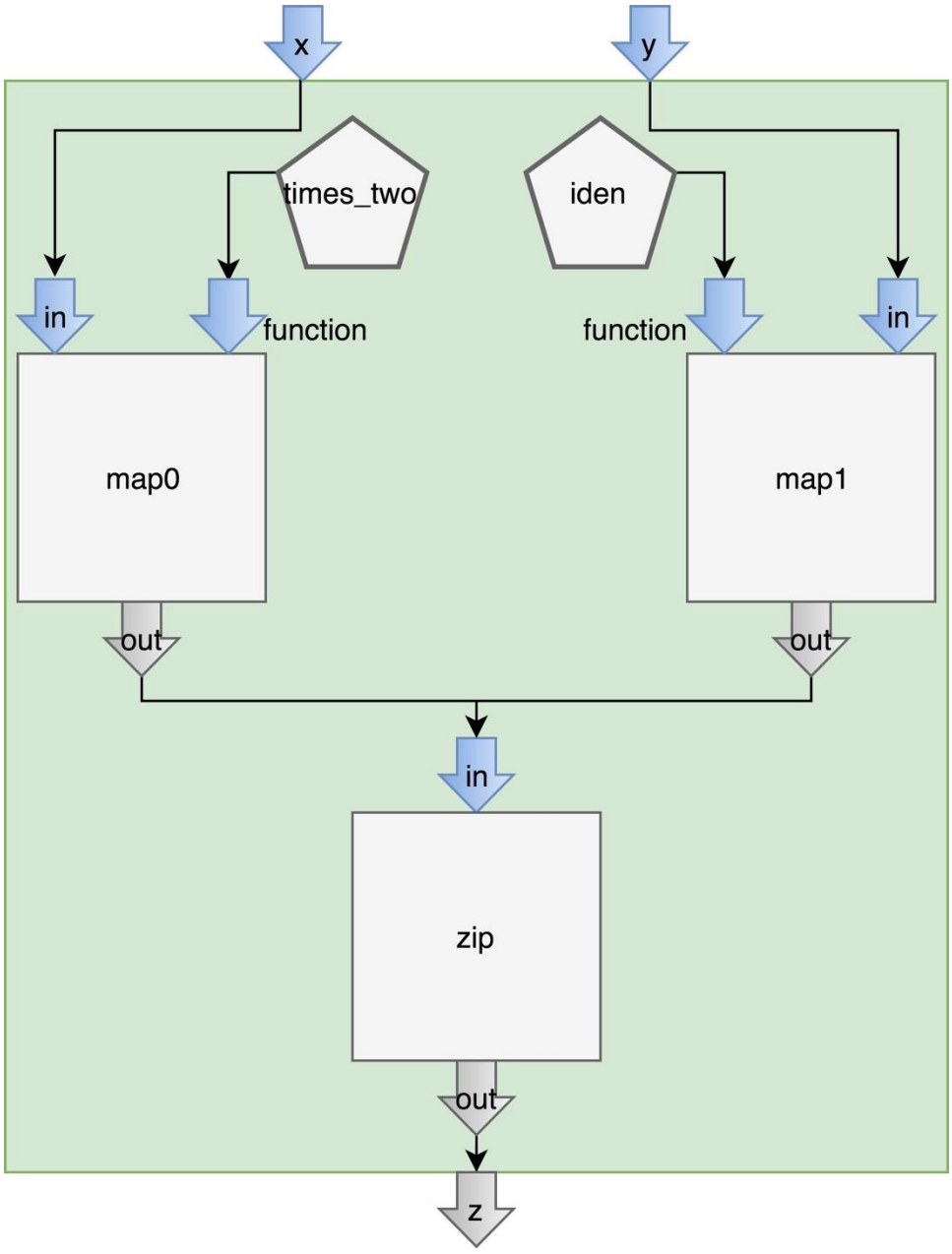

Figure 7: Example template. This is the same example as in Figure 6

Each template is assembled by connecting external and internal connections and then recursively assembling the subparts. For example, consider the template shown in Figure 7.

To assemble this template, we first create connections for external parameters. We connect the input *x* to the input *in* for **map0**, the input *y* to the input *in* for **map1**, and the output of **zip**, *out*, to *z*. We also connect the functions *times_two* and *iden*.

Next, we connect internal parameters. We connect each output of **map0** and **map1** to the input *in* of **zip**. Recall from the structure of the JSON that we specify indices for these connections: the input *in* of **zip** is a list of streams.

After connecting parameters, we recursively assemble the component templates. In this example, we would assemble the template for **map** and then for **zip**. The resulting component object is a streaming application with inputs and outputs.

To assemble a template, you can run the following command from a terminal:

```
IoTPy assemble template_name kwargs
```

template_name is the name of the template to assemble and *kwargs* is a list of keyword arguments for the template in the form of *arg=value*. These arguments are delimited by spaces. *kwargs* can be used to specify arguments that are primitive types. For example, you can specify integers, floats, strings, etc. You can also specify function names as strings in the form of *module_name.function*. The assemble framework will parse the string and look for the function under *modules/module_name*.

For example, to assemble the template for **meetup**, run:

```
IoTPy assemble meetup
```

To assemble a template with *kwargs*, run:

```
IoTPy assemble test window_size=1 function="test.iden"
```

You can also assemble templates by calling *IoTPy.tools.assemble* directly. This function takes the following parameters:

- *name*: the name of the part. This can be any string.
- *template_name*: the name of the template.
- *module_name*: the name of the module from which to load functions.
- *kwargs*: keyword arguments for inputs

We will discuss other optional parameters in the next section.

If you assemble templates by writing Python code, you can specify arbitrary objects as inputs to templates. You can also specify arbitrary functions that are not defined in modules/. *module_name* is the name of the module containing functions and will be the same as the module where you assemble templates. For example, consider the following code:

```
from IoTPy .tools import assemble
from IoTPy.code import Stream

def f(x):
    return x

x = Stream()
y = Stream()
assemble("test", "map", x=x, y=y, function=f)
```

This assembles the template for **map** and specifies that the function is *f*, where *f* is defined in this file. You can also specify imported functions, e.g. functions from NumPy [19]. This allows you to use functions from other modules without creating wrappers.

4.1 Shared memory

To assemble templates, we must create connections for external and internal parameters. For single process templates, we do this using shared memory. For each subpart, we create a dictionary that contains its keyword arguments. For example, we consider the connection [x, map1, in] in the template above. This specifies that the external parameter *x* connects to the input *in* for **map**. We create this connection by adding the value for *x* with key *in* to the dictionary for **map**. In Python code, this looks like:

```
subparts_dict["map1"]["in"] = dict_parts["x"]
```

In Python, each variable is a name that refers to a binding. Python does not have explicit pointers like other languages. Ideally, to connect internal parameters, we would want to assign the same pointer for each parameter so that they point to the same object in memory. For example, in the above template, to connect *out* from **map** to *in* for **zip**, we want the following:

```
map:out = x
zip:in[0] = x
```

where *x* points to an object.

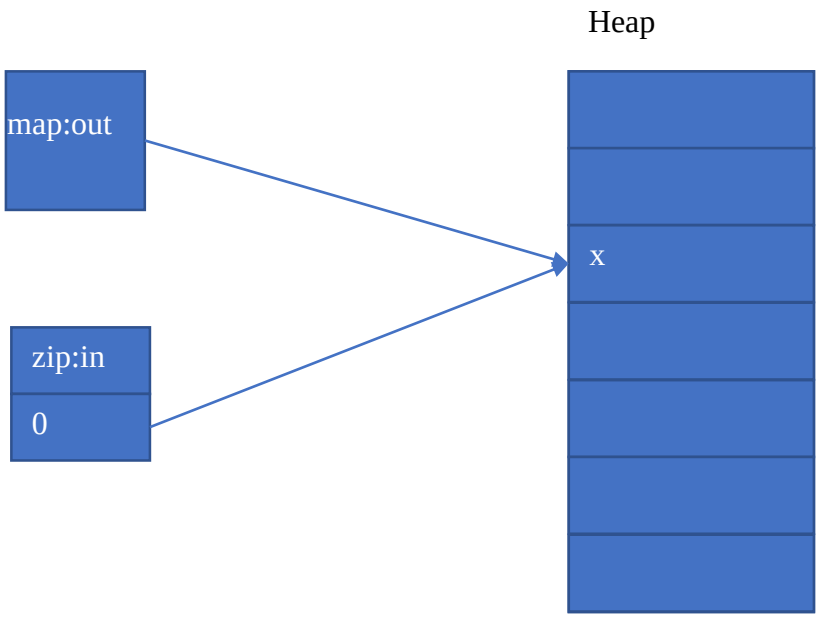

Figure 8: Example of subparts sharing stream

We could do this by doing the following:

```
x = Stream()
map:out = x
zip:in[0] = x
```

Indeed, this was the first approach taken to solve this problem. However, this has the constraint that we must know the type of *x* before assembling **map** or **zip**. In this example, the type of *x* is known, as it is a Stream. However, for templates that return arbitrary types, this does not work.

To do this in Python, we create a wrapper object called Value. This object has a member variable called *value*, which stores a value of any type. This is boxing the type. Then we do the following:

```
x = Value()
map:out = x
zip:in[0] = x
```

Now, both *map:out* and *zip:in[0]* are bound to the same object in memory. Then when we assemble **map**, we set the *value* member variable of *map:out* to Stream. This looks like the following:

```
map:out.value = Stream()
```

Then *zip:in[0].value* is the same Stream as *map:out*.

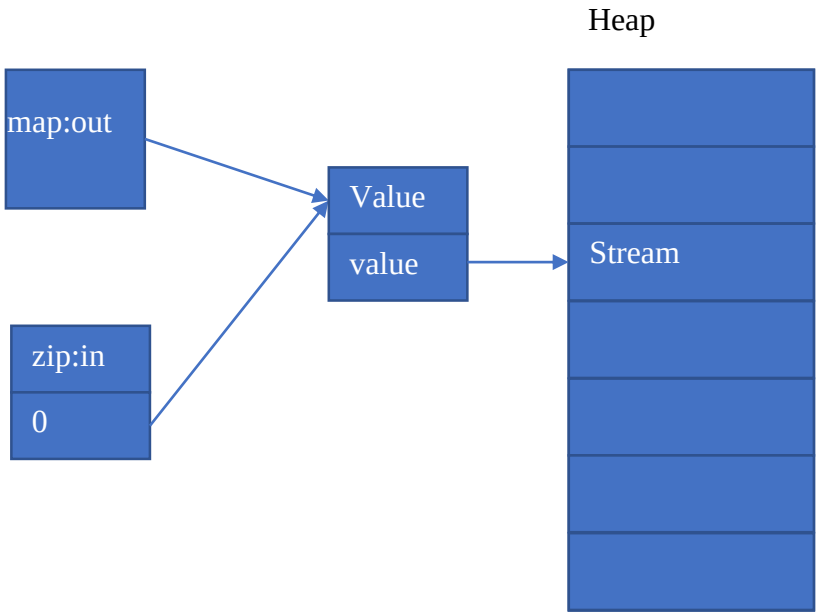

Figure 9: Using boxing to share streams

4.1.1 Typed parameters

When we create connections for internal parameters, we can create the object by looking at the parameter type. For example, to connect *map:out* to *zip:in[0]*, we set *map:out.value* = *Stream()*. This comes from the fact that *map:out* is defined as type Stream in the template. The two types we handle are:

- *stream*: We create a Stream() object
- *function*: We lookup the function from the module and import it

We can also handle parameters that are not typed. For example, consider the template for our Meetup example:

```
meetup() -> None
{
    out = source(function="ML.examples.meetup.source",
initial_state=0);
    m = func(function="ML.KMeans.KMeansStream",
parameters=[0, 0, 5]);
    train = attribute(in=m, attribute="train");
    model = train(in=out, function=train,
num_features=2, min_window_size=5, max_window_size=30,
step_size=2);
    plot(in=out, function="ML.examples.meetup.plot",
model=model, num_features=2, min_window_size=5,
max_window_size=100, step_size=2);
}
```

Let us take a look at the second line in the template:

```
m = func(function="ML.KMeans.KMeansStream",
parameters=[0, 0, 5]);
```

The **func** template runs a function and returns the result. In this case, we use it to construct an object of class KMeansStream(). Then *m* is an instance of this class.

In the next line, we specify that *attribute:in* is the same as *m*. However, we do not know the type of *m* before assembling the object. We handle this by setting *func.out* and *attribute:in* to the same Value object. Then when we assemble **func**, we set the value member of this object to the return value of KMeansStream().

```
x = Value()
func:out = x
attribute:in = x
```

Then:

```
func:out.value = KMeansStream()
```

This gives us the ability to conform to Python's loose typing and late binding in the template definition language.

5. Multiprocessing

We discussed how we assemble templates. This assembly is done in a single process. However, we can also assemble a template using multiprocessing. Multiprocessing enables us to run multiple threads simultaneously on a single machine, making a streaming application faster. Each thread runs on a separate core of the processor.

We implement multiprocessing by running each stream subpart in a template in its own process. By stream subpart, we mean a subpart that has stream inputs or outputs. For example, consider template in Figure 6.

We implement multiprocessing by running each map and zip in its own process. This allows us to run these concurrently.

To run a part with multiprocessing, we specify a multiprocessing flag in the command line:

```
IoTPy assemble test –multiprocessing
```

5.1 Implementation

Multiprocessing is implemented by creating queues for each process. Data between subparts is sent through queues. This is needed due to the fact that processes do not share memory. We use the multiprocessing module for Python to create processes and queues.

We first create a queue for the main template. This queue connects the external parameters to the subparts. We then create a queue for each subpart and create internal connections. Finally, we create sources and sinks for each queue. The sink takes in values on streams and appends them to the correct queue. The source reads values from the queue and adds them to streams.

The multiprocessing implementation for the template in Figure 6 is shown in Figure 10.

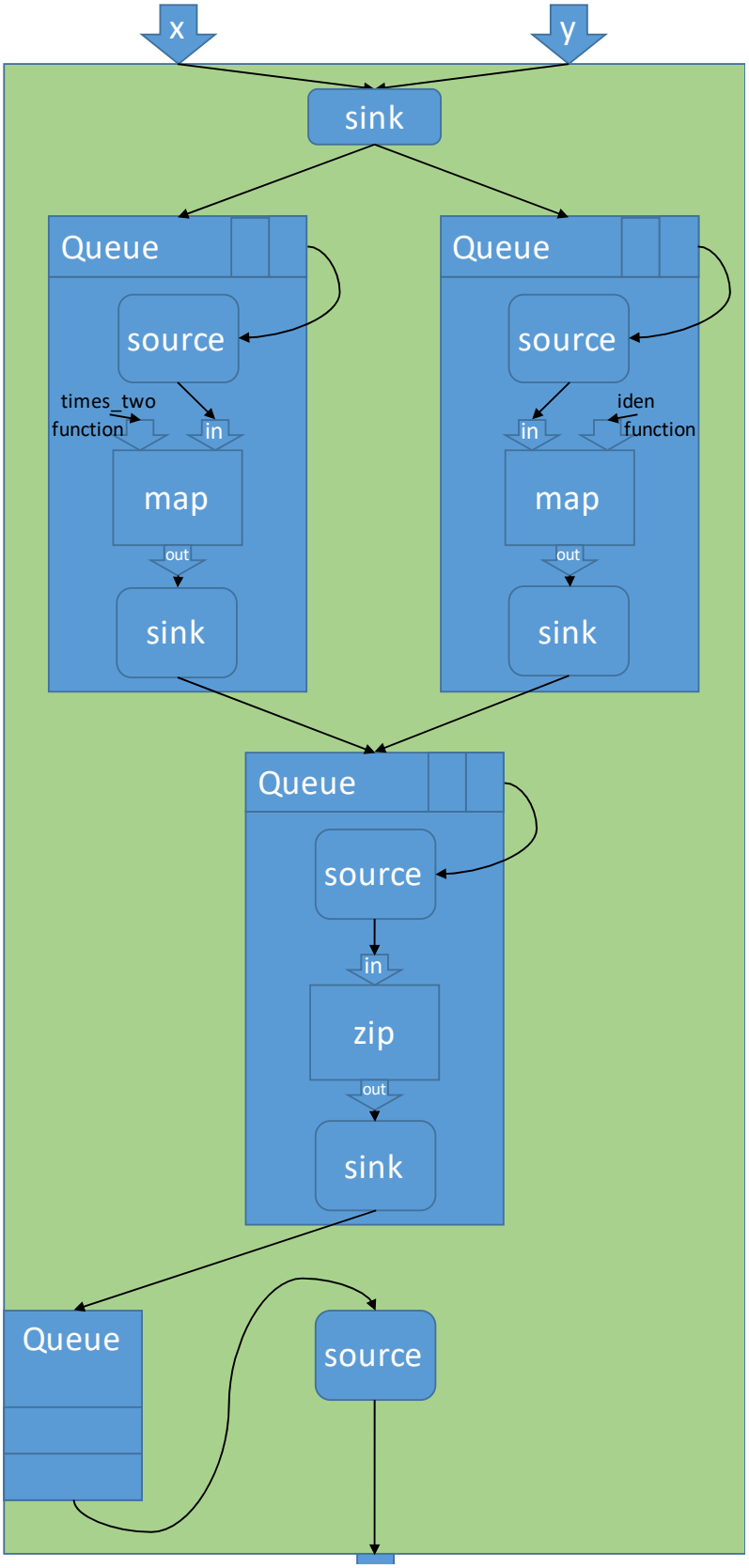

Figure 10: Multiprocessing for template in Figure 6

Values on *x* and *y* are read by a *sink* for the template, which appends them to the queues for each **map**. Each map has a *source* that reads values on its queue and adds it to the stream for **map**. Values from *map:out* then go to a *sink*, which appends them to the queue for **zip**. This continues and the values from **zip** go to the queue for the main template. Finally, the source for the main template reads these values and adds them to the stream *z*.

To do this, the parameters for each subpart need to have information about which queue they are sent to or come from. We assemble external and internal connections and set this information in the Value class. The Value class has member variables:

- *value*: stores a value of any type
- *dest*: list of destinations

For example, when we assemble the external connection *[x, map0, in]*, we do boxing of the input *x* in an instance of Value. We then set the *dest* member of this instance to *[map0:in]*. When we assemble the internal connection *[map0, out, zip, in, 0]*, we set the *dest* member of *map0:out* to *[zip:in[0]]*. After implementing all external and internal connections, we assemble the subparts in their own process.

Finally, we create a *sink* and *source* for the template. When the *sink* receives a value on *x*, it sends it to the queue for **map0** with a message that specifies that the value is for the stream *in*. The source receives the value on the queue and adds it to the correct stream.

We choose to run only stream subparts in processes as other parts are sequential. For example, consider the template for Meetup:

```
meetup() -> None
{
    out = source(function="ML.examples.meetup.source",
initial_state=0);
    m = func(function="ML.KMeans.KMeansStream",
parameters=[0, 0, 5]);
    train = attribute(in=m, attribute="train");
    model = train(in=out, function=train,
num_features=2, min_window_size=5, max_window_size=30,
step_size=2);
    plot(in=out, function="ML.examples.meetup.plot",
model=model, num_features=2, min_window_size=5,
max_window_size=100, step_size=2);
}
```

We cannot assemble **func** and **attribute** using multiprocessing as **attribute** depends on the output of **func** and **train** depends on the output of **attribute**. If we assemble these using multiprocessing, they will be concurrent and we cannot guarantee this order of execution.

6 Distributed Computing

In addition to multiprocessing, we can also distribute the execution of templates across many machines. This enables us to create a distributed system of stream processors that together form a distributed streaming application.

We distribute a template by sending stream subparts to different nodes, where a node refers to a machine in the distributed system. This is the same design as multiprocessing, except that a process that executes the subpart is replaced by a process on a different node.

This requires a distributed queuing infrastructure that allows readers and writers executing on remote machines. We use RabbitMQ [20] as a messaging broker to enable communication between subparts executing on different nodes. RabbitMQ uses Advanced Message Queuing Protocol (AMQP) [22]. RabbitMQ supports message exchanges and named channels within exchanges. We use these to create separate channels for commands and streaming data.

To distribute a template, we first create an exchange for the template in RabbitMQ. We then create external and internal connections as before. This is done with the *dest* values for each parameter. Then, we send each stream subpart to a specific RabbitMQ channel called "*assemble*". This is the command channel. All nodes listen on this channel and subparts are distributed using a round-robin algorithm by RabbitMQ. A node receives a subpart and then assembles it in a separate process. A node can also assemble a subpart using multiprocessing, thus enabling concurrent execution of a subpart on a node.

Each subpart has a queue that is bound to the exchange using the subpart's name as a routing key. The subpart has a *source* and *sink*. The *source* reads values on the queue and adds them to streams. The *sink* reads values on streams and sends them to the exchange with the routing key corresponding to the destinations. We see that the mechanism is identical to that used in multiprocessing except that we send values to the RabbitMQ exchange rather than to multiprocessing queues.

We show an example of a template distributed across four nodes. This is the same example as described in multiprocessing. The parts **map0**, **map1**, and **zip** are run on different nodes. The **example_part** template is run on the assembly node. The RabbitMQ server is hosted on a different node. We create an exchange for **example_part** and send values for different parts using the part name as the routing key.

To run a part with distributed computing, we specify a distributed flag in the command line:

```
IoTPy assemble test --distributed
```

We can also specify the *hostname*, *user*, and *password* for the RabbitMQ server using the --host, --user, and --password flags.

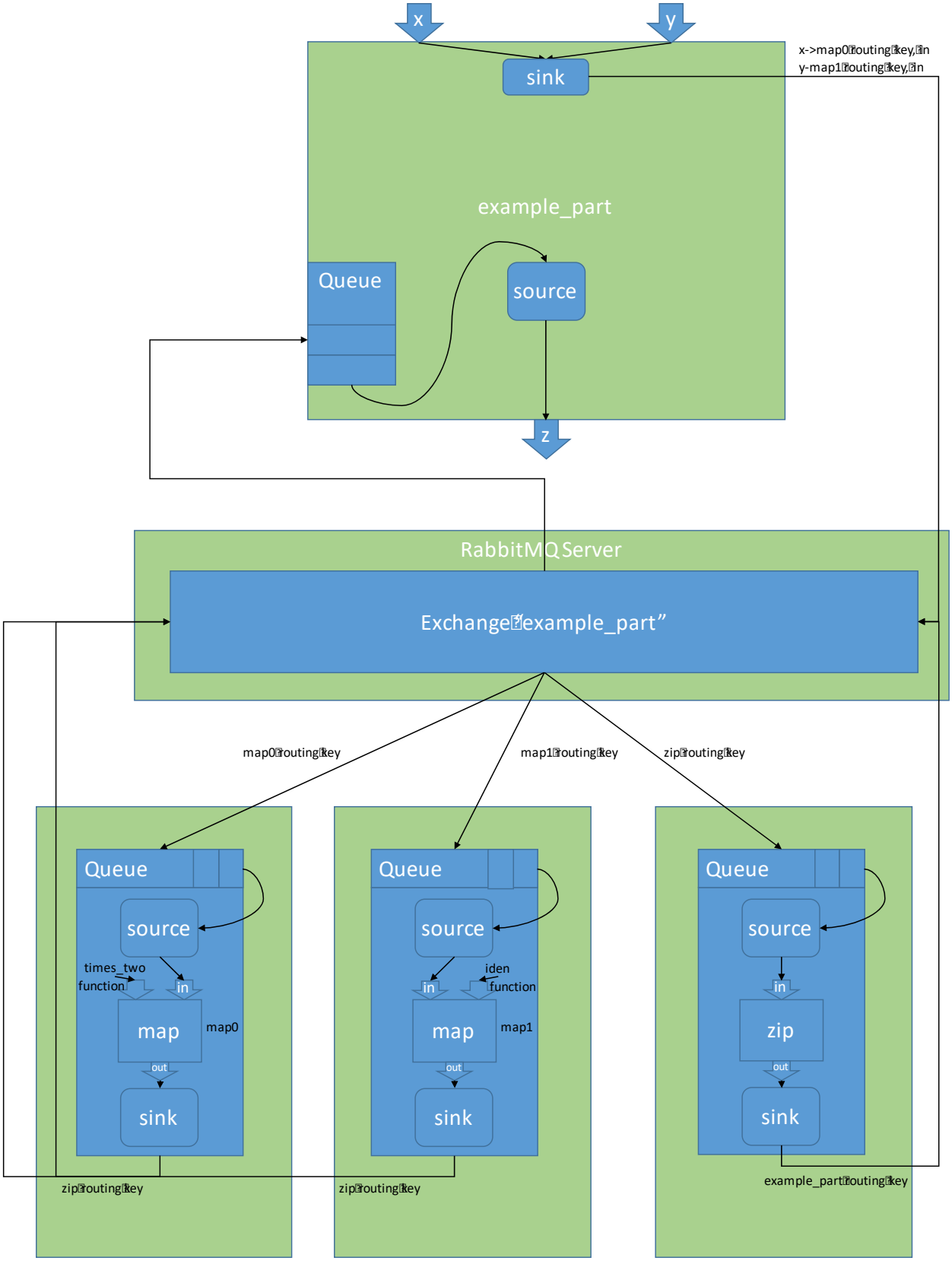

Figure 11: Distributed system for template in Figure 6 showing map0, map1, and zip running on different nodes.

7 Conclusion

The IoTPy system allows users to write streaming applications without requiring any ramp-up time. Non-programmers can use the template notation or UI to create templates by connecting other templates. Novice Python programmers can write functions that run on batch data at rest. Streaming experts can write streaming algorithms that are designed for streaming data and are more efficient. As an example of the extensibility of the system, we include a ML framework that allows users to write machine learning applications for streaming data. As can be seen from our example for the Meetup [7] application, we implemented a kmeans machine learning algorithm that was written for static data. We easily wrapped this algorithm and applied it in a plug-and-play manner to streaming data. This example shows that IoTPy allows the direct application of existing software development expertise to streaming data without a steep learning curve or understanding of streaming systems. We also include incremental algorithms that show the extensibility of IoTPy for various users, including experts.

Writing a streaming application using existing frameworks like Spark Streaming requires extensive knowledge and understanding of streaming systems and the specifics of the streaming framework. For example, the solution for Meetup implemented by Zelvenskiy [13] required Zelvenskiy to understand Spark's programming model and RDD construct, define multiple schemas, a receiver function, multiple transformation functions and many other utility functions. The amount of expertise required for Zelvenskiy's solution puts the streaming domain out of reach of novice users. On the other hand, our application which was similar to Zelvenskiy's was written in only 35 lines of simple code. Any novice programmer can write such applications; this demonstrates the power of the framework in making streaming ML accessible.

IoTPy also makes it simple to run streaming applications using multiprocessing or distributed computing. A user can run the same streaming application on a single process, multiple processes, or multiple machines by simply changing a configuration parameter without needing to change the template.

Users can contribute modules and templates to the system. We hope to build an extensible and easily discoverable streaming library that enables engineers and scientists to create reusable code for learning over streaming data, and plan to make the code usable by a large number of users.

We aim to add other languages to the system, such as C++ or Erlang. Users will be able to write streaming templates and run them in different languages. Since our distributed system is independent of the language, users can write distributed streaming applications that run on machines running different languages. This enables support for a variety of environments and operating systems.

References

1. Apache Hadoop. Retrieved from http://hadoop.apache.org/
2. Apache Spark - Lightning-Fast Cluster Computing. Retrieved from http://spark.apache.org/
3. Apache Storm. Retrieved from https://storm.apache.org/
4. Batty, M. (2013). Big data, smart cities and city planning. *Dialogues in Human Geography, 3*(3), 274-279.
5. Xia, F., Yang, L. T., Wang, L., & Vinel, A. (2012). Internet of things. *International Journal of Communication Systems, 25*(9), 1101.
6. Abu-Mostafa, Y. S., Magdon-Ismail, M., & Lin, H.-T. (2012). *Learning from data* (Vol. 4): AMLBook New York, NY, USA:.
7. Meetup: Extend your community. Retrieved from https://www.meetup.com/meetup_api/
8. Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., . . . Whittle, S. (2013). MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment, 6*(11), 1033-1044.
9. Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002). *Models and issues in data stream systems.* Paper presented at the Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.
10. Faulkner, M., Olson, M., Chandy, R., Krause, J., Chandy, K. M., & Krause, A. (2011). *The next big one: Detecting earthquakes and other rare events from community-based sensors.* Paper presented at the Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on.
11. Python Data Analysis Library. Retrieved from http://pandas.pydata.org/index.html
12. Scikit-Learn: Machine Learning in Python. Retrieved from http://scikit-learn.org/stable/index.html
13. Zelvenskiy, S. Spark Streaming, Machine Learning and meetup.com streaming API. Retrieved from http://www.slideshare.net/SergeyZelvenskiy/spark-streaming-45954549
14. JSON. Retrieved from http://json.org/
15. Extensible Markup Language (XML). Retrieved from https://www.w3.org/XML/
16. Twitter. Retrieved from https://twitter.com/
17. Pyparsing. Retrieved from http://pyparsing.wikispaces.com/
18. TinyDB. Retrieved from https://pypi.python.org/pypi/tinydb
19. NumPy. Retrieved from http://www.numpy.org/
20. RabbitMQ. Retrieved from http://www.rabbitmq.com/
21. Cugola, G., & Margara, A. (2010). TESLA: a formally defined event specification language. Paper presented at the Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, Cambridge, United Kingdom.
22. AMQP. Retrieved from http://www.amqp.org/