Building Scalable Services in Microsoft Azure with Python

This paper demonstrates how to use Microsoft Azure and the Python programming language to implement the compute nodes of a scalable scientific computing service. It uses Basic Local Alignment Search Tool (BLAST) queries as an example. BLAST is an application that searches biological databases to find regions of similarity between nucleotide or protein sequences. A BLAST search enables a researcher to compare a query sequence with a library or database of sequences, and identify library sequences that resemble the query sequence above a certain threshold.

The BLAST sample application in this paper is a scalable search service with an optional web front end. It uses a Microsoft Azure Service Bus Queue and Microsoft Azure Table service to communicate with the back-end BLAST compute nodes.

The back-end compute nodes can be either Microsoft Azure worker role instances that run on the Microsoft .NET Framework or Microsoft Azure virtual machines (VMs) that execute BLAST from a Python program. This paper describes how to create compute nodes that use Python. You can use the optional web front end that is based on the Microsoft .NET Framework to interact with the back-end compute nodes, or you can interact with back-end nodes by using Python.

A companion paper, *Scaling a Microsoft Azure Cloud Service*, describes the web front end and the worker instances that use the Microsoft .NET Framework.

Overview

Here is an overview of how the BLAST sample application works.

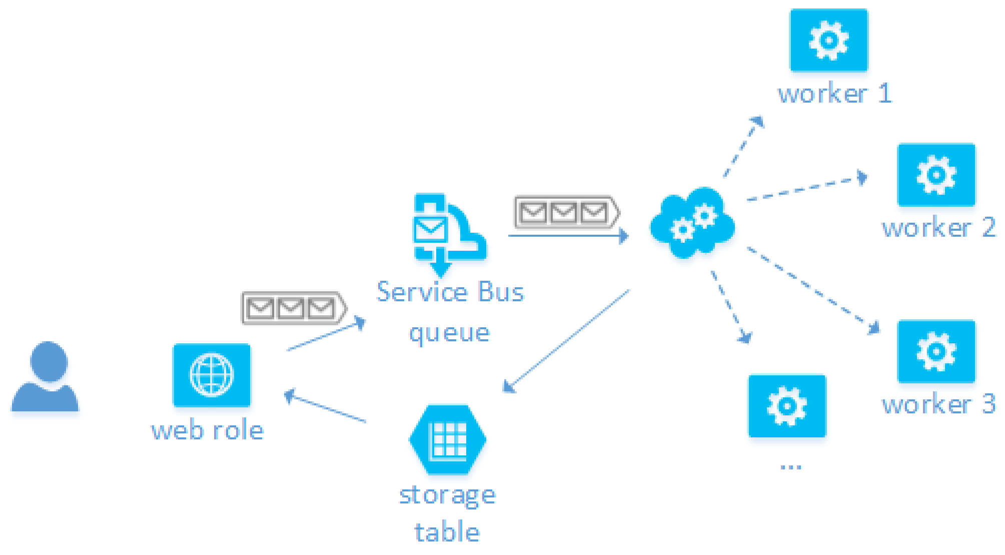

Figure 1: Diagram of BLAST sample application in action

The researcher enters queries in a web application that adds each new genomic query request from the user to a queue in the Microsoft Azure Service Bus. Microsoft Azure Service Bus is a service that lets you reliably pass messages between distributed components that run on the premises or in the cloud. One mechanism for communication is the Service Bus Queue, which is used by the BLAST example. A Service

Bus Queue offers first in, first out guaranteed message delivery and supports a range of standard protocols and APIs to put messages in the queue and to remove messages from the queue. (For more information about Service Bus, see the Service Bus documentation.)

The back end of the BLAST search application consists of compute nodes, which are also sometimes called worker nodes. This paper describes compute nodes that are realized by using a Python program that runs on Microsoft Azure VMs. You can configure the system to include as many VM instances as you need.

In the BLAST sample, idle compute nodes pull query requests from the Service Bus
Queue for processing. When the query is finished, the compute nodes record the results into Microsoft Azure Table service. Table service is a fault-tolerant, ISO 27001 certified NoSQL key/value store. Table service can be useful for applications that must store large amounts of non-relational data.

Configuring and Publishing the BLAST Application

Microsoft Azure provides convenient options for creating and managing the services it provides. You can perform these tasks with:

- The Microsoft Azure Management Portal
- A command-line interface
- Calls to the Microsoft Azure API from within the program
- The built-in automation features of Microsoft Visual Studio

To configure and publish the BLAST sample application, you will perform some steps from the Microsoft Azure Management Portal and some steps by using an SSH connection to the VM.

In the Microsoft Azure Management Portal, you create credentials that will allow your application to identify unique Microsoft Azure service endpoints and gain access to them. Then, if you want to use the web-based front end, you will set that up. Finally, you will use the Management Portal to create a Microsoft Azure VM and to publish the Python code to it in order to run the BLAST back end.

Using the Microsoft Azure Management Portal

In the Microsoft Azure Management Portal, you will:

- Create a Microsoft Azure Service Bus namespace that provides a unique name prefix for services such as queues
- Create a Service Bus Queue named **jobqueue**, if the queue doesn't already exits
- Create a Microsoft Azure storage account

These activities create credentials that will allow your application to identify unique Microsoft Azure service endpoints and gain access to them. You can find detailed information for performing these steps in the “Using the Management Portal” section of the companion article, *Installing the Microsoft Azure BLAST Example*.

Configuring BLAST.NET (Optional)

If you want to use the .NET-based web front end of the BLAST sample application, you can find detailed information on how to configure and publish it in the companion article, *Installing the Microsoft Azure*

BLAST Example. Follow the instructions in the sections of that paper that are called "Configuring the Environments" and "Publishing the BLAST Demo."

If you decide not to configure the .NET version of this BLAST sample application, you can use the instructions later in this paper for interacting with the back end server by using Python.

Configuring the Python-Based Back End

This section describes the steps you need to take to configure and publish the Python back end.

Task 1: Create a VM

Use the Management Portal to create an Ubuntu Server 12.04. If you need help creating the VM, see the paper, *Microsoft Azure for Linux and Mac OS Users* for help.

Task 2: Copy the scripts

1. After you have created the VM, create a directory named **/Source** for your Python BLAST code.
2. Download the Python BLAST code from the github repository at https://github.com/MSRConnections/Azure4Research-TechnicalPapers. The Python BLAST code is found in the **Azure4Research-TechnicalPapers** repository under **Building_Scalable_Services_in_Windows_Azure_with_Python_BLAST/Source**.
3. Copy the Python BLAST code to your **/Source** directory on the VM that you created in Task 1. You can use **wget**, **curl,** or **git clone** to perform the copy from github.

   For example, to download the repository by using **git clone**, type the following commands:

   ```
   sudo apt-get install git
   git clone https://github.com/MSRConnections/Azure4Research-TechnicalPapers.git
   ```

Task 3: Install the application

You can run the install procedure from any directory on the VM. The instructions below assume that you are logged in as the default user for a Microsoft Azure VM (**azureuser**) and that you are installing from your home directory. The instructions also assume that you have downloaded the Python BLAST code to a directory named **/Source** and that you want to install the application in the **/home/azureuser/BLAST** directory.

To do a full install, follow these steps:

1. Download and unzip the setup and run-time files from https://github.com/MSRConnections/Azure4Research-TechnicalPapers/archive/master.zip.

2. Copy all of the setup files from Building_Scalable_Services_in_Windows_Azure_with_Python_BLAST**/Source/util** to the **/home/azureuser** directory. Copy all the run-time files from Building_Scalable_Services_in_Windows_Azure_with_Python_BLAST**/Source/src** to the **/home/azureuser/BLAST** directory.

3. The sample stores its configuration settings in the **blast_config.py** file. You should review these settings and make changes if necessary. Here is the configuration code.

   ```
   import os
   ```

```
database_root_path = "/home/azureuser/BLAST/blastdb"
ncbi = os.path.join(database_root_path, "ncbi")
est_human = os.path.join(ncbi, "est_human")
inputncbi = os.path.join(database_root_path, "inputncbi")
blastout_root = os.path.join(database_root_path, "../blastout")
blast_exe = "/home/azureuser/ncbi-blast-2.2.28+/bin/blastn"
blast_output_file_template = "%s.out"
blast_input_file_template = "input_%s"
```

4. From **/home/azureuser**, run ***source*** **download_and_install_all.sh** to download BLAST, unpack it, and install all the Python and Microsoft Azure dependencies***.***

5. Credentials and the identities of Microsoft Azure resources are found in the **azure_config.py** file in the **BLAST** directory. You will need to replace the values for the service bus and storage account credentials with the values you created earlier by using the procedures given in the paper, *Installing the Microsoft Azure BLAST Example*. Here is the code that contains the credentials and identities of the Microsoft Azure resources.

```
# Storage
azure_blast_data_storage_account_name = 'blastfileseu'
azure_blast_data_storage_account_key = 'xkTjP…x7kUPqxbyX6Q=='

azure_storage_account_name = 'blaststore'
azure_storage_account_key = '99…q71/Q=='

history_table_name = 'blastlog'
job_table_name = 'searchtask'

# Service Bus
service_bus_namespace = 'sb-azure4research-blast'
service_bus_key = 'aC7Hfbv….UZZ8='

request_queue_name = 'jobqueue'
```

6. To start the application, run the following commands in a shell:

```
cd /home/azureuser/BLAST

python master_blaster.py
```

These commands start a non-terminating processing loop that waits for search requests to appear in the queue. See "The Top-Level Processing Loop" section later in this paper for more information.

Note: You can also run the **master_blaster.py** program as a daemon but running as a daemon is not required.

The rest of this article gives an overview of how Microsoft Azure services work together to perform BLAST queries once the application is running. The sections discuss:

- Adding requests to the queue
- The top-level processing loop
- Monitoring the queue for new requests
- Storing the results of the BLAST search in Microsoft Azure Table service
- Initializing the worker role with data from Microsoft Azure Blob service

Adding a New Request to the Queue

If you installed the web front end, you can create BLAST queries in your web browser.

If you did not install the web front end, you can use Python to create queries. You can send requests to BLAST by using the program **send_blast_request.py**. You can request a specific number of queries (for example, `python send_blast_request.py 42` requests **input_42**). If no specific number is passed, random inputs that range from 1 to 200 are submitted. To view the recent results that have been saved to Table service, use the command, **python show_recent_blast_logs.py**.

The Top-Level Processing Loop

After the query request is added to the queue, it remains in the queue until one of the worker instances retrieves it for processing. The Service Bus delivers messages to one recipient at a time. A message may be delivered more than once if a previous recipient fails before marking the entry as successfully processed.

The top-level Python file **master_blaster.py** contains the logic of the compute node's top-level loop. The file starts with statements that import the project modules that the compute node will use. Here is the code.

```
import datetime
import json
from azure_config import *
from blast_config import *

from download_blast_database import make_sure_blast_database_is_downloaded
from run_blast_command import run_blast
from upload_to_blast_viewer import upload_to_blast_viewer
from get_next_blast_request import *
from log_blast_result import *
```

Note that importing **from azure_config** loads the Microsoft Azure-specific configuration keys for both Service Bus and Microsoft Azure Storage.

After the modules are imported, the main logic begins. Here is the code.

```
make_sure_blast_database_is_downloaded()

while True:
    try:
        print('About to access queue (@ %s)...' % now())
        msg = get_next_blast_request(timeout_seconds)
        print('get_next_blast_request returned (@ %s)' % now())
        if msg.body == None:
            print("No message returned (timeout)")
            print("Sleeping for %d seconds" % sleep_seconds)
            time.sleep(sleep_seconds)
        else:
            print("Message body = %s" % msg.body)
            blast_job = json.loads(msg.body)
            input_file = blast_job['InputFile']
            if (not input_file is None):
                print('run_blast(%s)' % input_file)
                output_file_path = run_blast(input_file)
                print('uploading "%s" to blast viewer' % output_file_path)
                hash = upload_to_blast_viewer(output_file_path)
                url = 'http://bov.bioinfo.cas.unt.edu/cgi-bin/viewhits.cgi?hash=%s' %
                                                                                  hash
                print('url = %s' % hash)
                blast_job['Hash'] = hash
                record_blast_job(blast_job) # so it shows up in BLAST.NET UI
                log_blast_result(input_file, url, blast_job['Id'])
                mark_blast_request_completed(msg)
    except:
        print
        print "EXCEPTION WAS RAISED... something didn't work that time.. trying again"
        print
```

The call to the **make_sure_blast_database_is_downloaded** function is important because it ensures that the BLAST database is downloaded from blob storage and stored locally on this node. The database is about 1.5 GB in size. See "Storing Data in Microsoft Azure Blob Service" later in this paper for more information.

The **while True:** loop runs until it is interrupted. The body of the loop calls the **get_next_blast_request** function to see if there is a message on the Service Bus Queue. If there is no message, the program pauses, then tries again. See "Monitoring the Queue for New Requests" later in this paper for more information.

If a message is available, it is processed in three steps.

1. The **run_blast** function executes the query on the local VM.

2. The **upload_to_blast_viewer** function uploads the query output result to the BLAST viewer site hosted at University of North Texas (unt.edu). Results uploaded to the BLAST viewer site are saved for 60 days. They can be found by using the special hash identifier returned by the site after a successful upload. This hash is used to construct a URL for direct access.

3. The **log_blast_result** function stores the URL where you can find the results, along with its expiration date, in Table service. See "Storing Data in Microsoft Azure Table Service" later in this paper for more information.

Monitoring the Queue for New Requests

The BLAST application uses the Microsoft Azure SDK for Python to pull messages from the Microsoft Azure Service Bus Queue. Here is the code for getting the BLAST request.

```
from azure.servicebus import *
import json
import uuid
from azure.servicebus import *
from blast_config import *
from azure_config import *
from ticks import *

sbs = ServiceBusService(service_bus_namespace,
                        service_bus_key,
                        'owner')

sbs.create_queue(request_queue_name)

peek_lock = False

def get_next_blast_request(timeout_seconds = 30):
    msg = sbs.receive_queue_message(request_queue_name, peek_lock, timeout_seconds)
    return msg
```

The code calls the **receive_queue_message** method of the **ServiceBusService** class, which is provided by the Microsoft Azure SDK for Python.

The **peek_lock** method determines whether a message will be delivered more than once—if a previous recipient fails—before marking the entry as successfully processed. If the value of **peek_lock** is **false**, then Service Bus returns the next message in the queue and deletes it all in one operation. When **peek_lock** is **true**, the message can re-appear on the queue for reprocessing if it is not deleted within a specific window.

Storing Data in Microsoft Azure Table Service

In order for users to monitor BLAST job requests, the BLAST sample keeps a running log of recent BLAST results by storing the links to the BLAST Viewer site in a Microsoft Azure Table. This table also includes the number of the input file used, the direct URL for viewing this result with the graphical viewer on the website, and the expiration for the URL (the host site allows them to stay for 60 days). Because BLAST results are non-relational data, the data can be efficiently managed in a key/value store. The following code shows how to create and use table storage.

```
import datetime
from ticks import *
from azure.storage import TableService
from blast_config import *
from azure_config import *

ts = TableService(azure_storage_account_name, azure_storage_account_key)
ts.create_table(history_table_name)

def calc_expiration():
    today = datetime.datetime.utcnow()
    expiration = today + datetime.timedelta(days=60)
    expiration = str(expiration)
    return expiration

def log_blast_result(input_file, url, id):
    partition_key = str(ticks_since_epoch())
    print('partition_key = %s' % partition_key)
    expiration_date = calc_expiration()
    ts.insert_entity(
        history_table_name,
        {
            'PartitionKey' : partition_key,
            'RowKey': input_file,
            'Id': id,
            'Url': url,
            'UrlExpiration': expiration_date
        }
    )
```

The **log_blast_result** function inserts a new value into table storage. The **PartitionKey** is a timestamp. The **RowKey** is the file name used to identify the BLAST input file. In the sample input set that ships with BLAST, these are values 1 to 200. Microsoft Azure requires that both **PartitionKey** and **RowKey** be stored as strings. The **Url** and **UrlExpiration** properties describe the URL from which the results can be accessed by using the BLAST Viewer.

Storing Data in Microsoft Azure Blob Service

A BLAST search involves reading a set of large files. To perform efficient searches, each worker role instance needs a local copy of the genomic database used by BLAST.

The BLAST sample handles this requirement by storing the files in a central location by using Microsoft Azure Blob service. The files that are stored in Blob service are considered the master copy.

When a compute node begins processing its first query, it copies the BLAST file set from the master copy and saves it locally. As a result, newly deployed instances have access to the same files as other running instances. The copy operation only occurs one time for each worker instance. Here is the code for downloading from blob storage.

```
import os, platform, base64
from azure.storage import *
from blast_config import *
from azure_config import *

# ...

blob_service = BlobService(account_name=azure_blast_data_storage_account_name,
account_key=azure_blast_data_storage_account_key)
blob_container_names = ['inputncbi', 'ncbi']
blob_chunk_size = 4 * 1024 * 1024

def make_sure_path_exists(path):
    if not os.path.exists(path):
        os.makedirs(path)

def is_local(file_path, expected_size):
    if not os.path.exists(file_path):
        return False
    else:
        return os.path.getsize(file_path) >= expected_size

# python download methods modeled on:
# http://azure.microsoft.com/en-us/documentation/articles/storage-python-how-to-use-blob-
storage/
def download_large_blob(blob_service, blob_container_name, blob_name, file_path):
    props = blob_service.get_blob_properties(blob_container_name, blob_name)
    blob_size = int(props['content-length']) # same value as
                                             # blob.properties.content_length
                                             # but we don't have a blob handy


    index = 0
    with open(file_path, 'wb') as f:
        while index < blob_size:
            chunk_range = 'bytes={}-{}'.format(index, index + blob_chunk_size - 1)
            data = blob_service.get_blob(blob_container_name, blob_name,
                                               x_ms_range=chunk_range)
            length = len(data)
            index += length
            if length > 0:
                f.write(data)
                if length < blob_chunk_size:
                    break
            else:
                break

def download_small_blob(blob_service, blob_container_name, blob_name, file_path):
        stream = blob_service.get_blob(blob_container_name, blob_name)
        with open(file_path, 'w') as f:
            f.write(stream)

def download_if_needed(blob_container_name, blob, directory_path):
    file_path = os.path.join(directory_path, blob.name)
    expected_size = blob.properties.content_length

    if not is_local(file_path, expected_size):
        if expected_size < blob_chunk_size:
            download_small_blob(blob_service, blob_container_name, blob.name, file_path)
        else:
            download_large_blob(blob_service, blob_container_name, blob.name, file_path)
    else:
        print(' ... skipping download of %d byte %s -- local copy found' %
                                                   (expected_size, file_path))

def download_blast_database(blob_service, database_root_path):
    make_sure_path_exists(database_root_path)

    for blob_container_name in blob_container_names:
        directory_path = os.path.join(database_root_path, blob_container_name)
        make_sure_path_exists(directory_path)

        print(blob_container_name)
        blobs = blob_service.list_blobs(blob_container_name)
        for blob in blobs:
            download_if_needed(blob_container_name, blob, directory_path)

def make_sure_blast_database_is_downloaded():
    download_blast_database(blob_service, database_root_path)
    make_sure_path_exists(os.path.join(database_root_path, '../blastout/'))
```

The **BlobService** class gives access to Microsoft Azure blob operations. The **get_blob** method of the Microsoft Azure SDK for Python is used to retrieve data from blob storage.

Summary of Websites

Here is a list of the websites listed in this article.

To download and install the BLAST sample described in this paper, see *Installing the Microsoft Azure BLAST Sample* (http://research.microsoft.com/en-us/projects/azure/installing-the-windows-azure-blast-example.pdf).

To learn about the Microsoft .NET Framework front end, see *Scaling a Microsoft Azure Cloud Service* (http://research.microsoft.com/en-us/projects/azure/scaling-a-windows-azure-cloud-service.pdf).

For information about Microsoft Azure Service Bus, see the Service Bus documentation. (http://azure.microsoft.com/en-us/documentation/services/service-bus/)

For more information about BLAST, go to the BLAST home page. (http://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastHome)

For more information about developing, deploying, and managing Python applications for Microsoft Azure, download the Microsoft Azure SDK for Python. (http://azure.microsoft.com/en-us/documentation/articles/python-how-to-install/)