Apache Hama Design Document v0.5

Introduction
Hama Architecture
    BSPMaster
    GroomServer
    Zookeeper
BSP Task Execution
    Job Submission
    Job and Task Scheduling
    Task Execution Lifecycle
Synchronization
    Fault Detection / Fault Tolerance
Future Work
References

Introduction

Apache Hama is a distributed computing framework based on BSP (Bulk Synchronous Parallel) [1] computing technique for massive scientific computations (e.g., matrix, graph, network, ..., etc) designed to run on massive datasets stored in Hadoop Distributed File System(HDFS). It is currently an incubator project by the Apache Software Foundation.

Apache Hama leverages BSP computing techniques to speed up iteration loops during the iterative process that requires several passes of messages before the final processed output is available. It provides an easy and flexible programming model, as compared with traditional models of Message Passing [2].It is compatible with any distributed storage (e.g., HDFS, HBase, …, etc), so you can use the Hama BSP on your existing Hadoop clusters. Finding shortest paths, value of PI etc. are some of the problems tackled by Hama today.(See http://wiki.apache.org/hama/Benchmarks). for "Random Communication Benchmark" results) The document covers the following in detail:

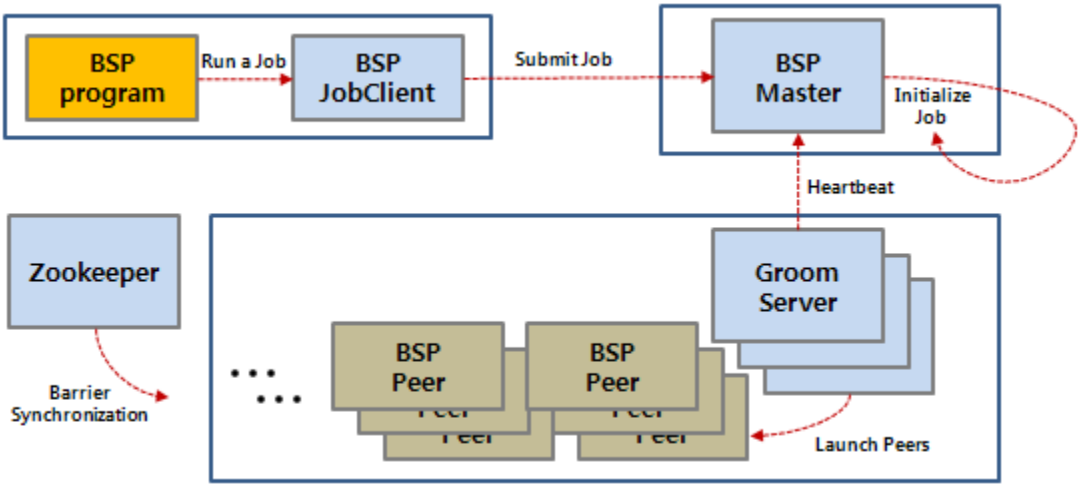
Hama Architecture:
The document explains the design of the whole system and its sub-components. It explains the role of each components and the part it plays for successful execution of tasks.
BSP Job Lifecycle:
Once we familiarize ourselves with the infrastructure, the document explains how the jobs are executed from the submission process till its completion.
Future plans:
Then the document explains what is to be expected from the system in the future. This is a good place for interested people to work on in the future and contribute to Hama.

Related Documents: Please refer to the Hama Installation Guide for installing Hama on a cluster. There is also a document on Hama Programming Model that explains the BSP model in detail for implementing different algorithm

If you have any questions on the content of the document, feel free to ask questions on our mailing thread - **hama-user@incubator.apache.org**

Hama Architecture

Architecture of Hama is very similar to Hadoop architecture, except for the portion of communication and synchronization mechanisms. The figure below gives a gist of Hama architecture. In the coming sections we would be going into each part of the figure in detail.



Hama consists of three major components: *BSPMaster*, *Groom Servers* and *Zookeeper*.

BSPMaster

BSPMaster is responsible for the following:

- Maintaining groom server status.
- Controlling super steps in a cluster.
- Maintaining job progress information.
- Scheduling Jobs and Assigning tasks to groom servers
- Disseminating execution class across groom servers.
- Controlling fault.
- Providing users with the cluster control interface.

From the responsibilities, we can A BSP Master and multiple grooms are started by the script. Then, the BSP master starts up with a RPC server for groom servers. Groom servers start up with a BSPPeer instance and a RPC proxy to contact the BSP master. After it's started, each groom periodically sends a heartbeat message that encloses its groom server status, including maximum task capacity, unused memory, and so on.

Each time the BSP master receives a heartbeat message, it brings the groom server status up-to-date. Then, the BSP master makes use of the groom servers' status in order to effectively assign tasks to idle groom servers and returns a heartbeat response that contains assigned tasks and others actions that a groom server has to do. For now, we have a FIFO [3] job scheduler and very simple task assignment algorithms.

GroomServer

A Groom Server (shortly referred to as groom) is a process that performs BSP tasks assigned by the BSPMaster. Each groom contacts the BSPMaster and takes assigned tasks and reports its status by means of periodical piggybacks with the BSPMaster. Each groom is designed to run with HDFS or other distributed storages. Basically, a groom server and a data node should be run on one physical node to provide the best performance.

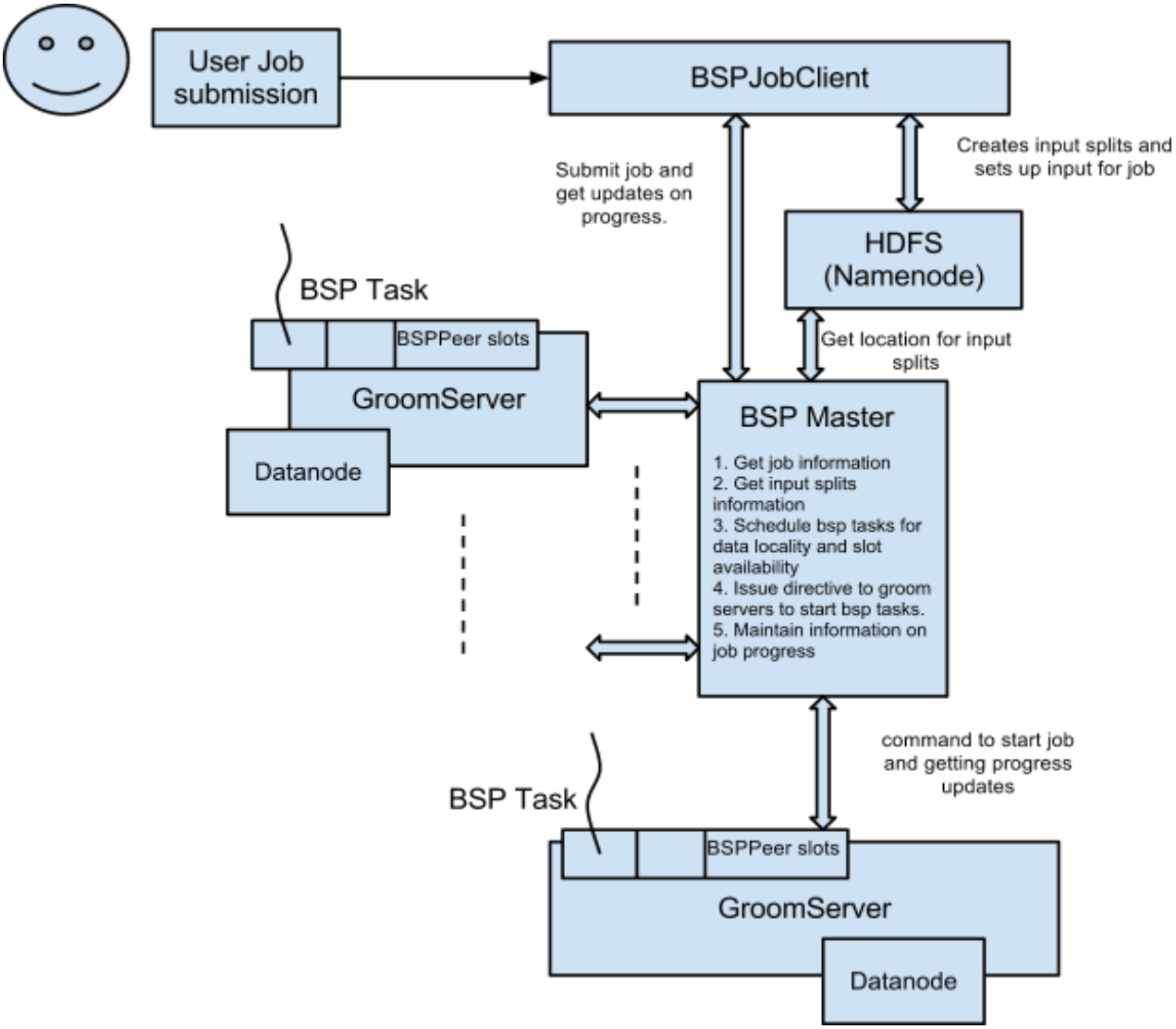
Zookeeper

A Zookeeper is used to manage the efficient barrier synchronisation of the BSPPeers. (Later, it will also be used for the area of a fault tolerance system)
Zookeeper is basically launched parallel to the BSPMaster and keeps track of various additional information. Each peer is connected to Zookeeper, as well as all the tasks.

BSP Task Execution

In this section, we shall see how Hama Framework handles the jobs assigned to it by users. In the process we would also understand the lifecycle of tasks and the actions taken by the framework behind the curtains.

Job Submission



The diagram above shows how a job submitted is handled by Hama framework. When a user submits a job, the BSPJobClient does the following for the user:

- Establishes a communication channel with BSP Master.
- Writes the required jar files for the job to run on Hama's filesystem
- Create splits for input and partitions (if defined)
- Submit the job for execution to BSP Master

- Periodically update the user with the status of the job.

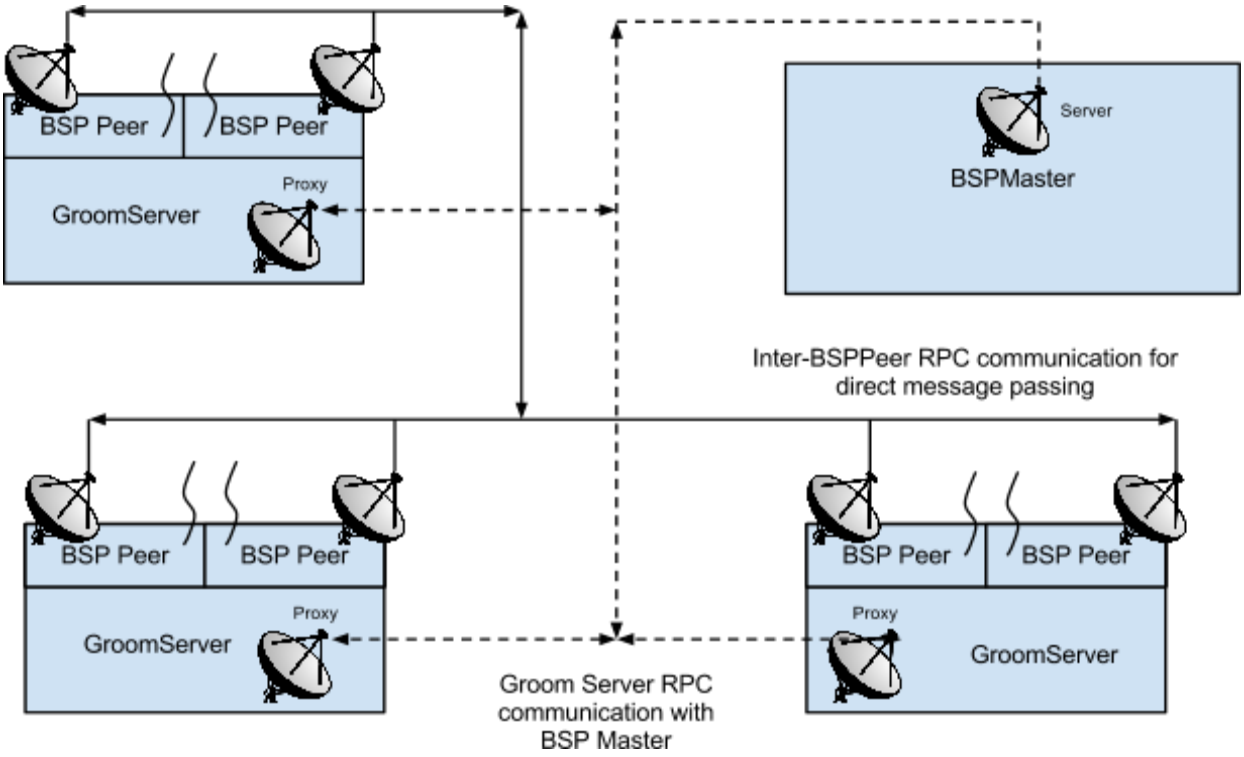
BSPJobClient uses Hadoop's RPC framework to communicate with BSP Master. It creates entries in the BSP system directory, writes the jar files and required configuration files as XML for the job. Based on the number of partitions specified in the job configuration or from the inputs, it creates input split information for the job. All this information is read by the BSP Master to schedule jobs for the user. The BSPJobClient updates itself with the progress(read current superstep counts) of the job and periodically informs user on the same. It also notifies the user if their job fails.

Job and Task Scheduling

When BSPJobClient submits a request to schedule a new job, the BSP Master infers the resources required for the job to complete from the job configuration submitted. It looks into the existent groom membership information, slots available to run tasks per groom server as well as the location of input splits in the cluster. The job-scheduler initializes the tasks for the job. It makes a best effort to schedule every task such that it runs on the machine where its respective input split is located. The tasks for which data-locality could not be achieved are scheduled on Groom Servers have slots for task to run. Once the BSP Master has found slots for scheduling all the tasks, it sends directive to the concerned groom servers to start new tasks inside a BSPPeer instance. The BSPPeer constantly notifies the BSPMaster over RPC communication on the progress of the job.

Task Execution Lifecycle

Thus, BSP Master would initiate a bunch of BSP Tasks on several Groom Servers. The tasks execute in a synchronous manner along with each other. It executes a series of supersteps. Between each superstep, all tasks belonging to a job enters a synchronization barrier using the Zookeeper service. In most cases, the tasks read their inputs from the filesystem, database or other source of data in the first superstep. The tasks share their intermediate computed values among each other directly by sending messages to each other directly before entering the synchronization period. BSP Master is oblivious of this communication. It only maintains the peer progress information periodically using a separate RPC communication infrastructure. The diagram below shows the communication infrastructure design used by tasks in Hama. All the tasks send messages to each other and execute all their supersteps. A task, once assigned to a Groom Server, continues its execution until the last superstep is executed. It does not get re-scheduled on a different Groom Server. In the event of failure, the job is marked as failed and get killed. Work is in progress for fault tolerance in Hama.



Synchronization

We currently using Zookeeper as the synchronization service. Zookeeper is powering large scale clusters and its synchronization primitives are very simple to use and have low overhead. Hama internally uses a double barrier synchronization, which first barrier syncs at the beginning of a communication step and once again barrier syncs when all the messages have been sent and received. All this is hidden behind a Synchronization Service keeping the API as simple and high level as it should be for a barrier synchronization.

However there are a lot of other types of barrier synchronization implementations and we constantly want to improve this implementation to lower the global overall cost as well as the latency involved.

Fault Detection / Fault Tolerance

Currently we are able to detect failure and have methods to save stateful data between the superstep (checkpointing of messages), but we think that this is not enough to provide a full fault-tolerant version of BSP in its current programming model.
We are working on this issue for a while now to provide a solution that keeps the open interface of BSP and hides the fault tolerance in our framework to make implementation and production use of Hama on commodity hardware very easy.

Future Work

Our future plans focus on providing a good and robust fault tolerance feature in the near future. While constantly improving the efficiency of the framework, e.G. improving performance, memory usage, improving overall scalability.
Currently Hama can be used as a batch computation framework, but we see uprising interest in using it as a real-time system. We want to enable and improve both parts of this framework.

Of course we want to add new features to Hama as well, for example the access to remote BSPPeer memory.

On the algorithmic part of Hama we want to add a various amount of algorithms ranging from graph theory and maths to machine learning and data mining applications.
In the math package we want to include efficient matrix multiplication and transformation algorithms, whereas we want to add constantly new useful graph examples.
We also see forward to have graph examples that can not be expressed with the Pregel API forced in frameworks like Giraph or GoldenOrb, rather than needing some kind of advanced supervision of the graph which is only available at the low level BSP primitives.

On the machine learning side, we are working on widely ranged algorithms with researchers around the globe. Unsupervised learning algorithms like K-Means Clustering already have been programmed and used with superior performance [1].
We are looking forward to see parallel SVM (Support vector machine) and ANN (artificial neural network), as well as DBSCAN (**d**ensity **b**ased **s**patial **c**lustering of **a**pplications with **n**oise) implementations with BSP soon.

References

[1] http://wiki.apache.org/hama/Benchmarks#K-Means_Clustering