# Gopher Architecture

This document describes the Gopher Design and its Architecture.

## High Level Architecture

Gopher is a programming framework for graph analytics. It’s a distributed framework which enables running analytics on large graphs which can be deployed in clusters and cloud. Gopher complements [google pregel](http://dl.acm.org/citation.cfm?id=1582723) model but model but relaxes the constraint on vertex-centric programming that forces many unnecessary supersteps and the added coordination overhead. Rather, Gopher takes a subgraph-centric notion where the user's logic has access to the entire subgraph structure, its state and the remote edges (rather than a single vertex and its edges). This allows more processing to take place within a subgraph before coordinating across subgraphs using synchronized message-passing across supersteps. Furthermore, Gopher exposes not just a single subgraph to the user logic but also allows them to bind to GoFS so that the user logic can access an iterator over subgraphs. This offer a hybrid programming model that leverages the best features of MapReduce (iterator over subgraphs, rather than tuples) and Pregel (subgraph, rather than vertex, centric computation with BSP-style message passing). Alternatively, users can use/implement their own backend storage model to pass subgraph iterators to Gopher instead of GoFS. Gopher is implemented on top of the [award-winning](http://www.cloudbus.org/ccgrid2012/cfp-scale.html) ***Floe***continuous dataflow engine for composable analytics.

Gopher is a Floe workflow which implements the Bulk synchronous parallel model. Gopher provides subgraph centric programming abstractions to the user where users can implement their subgraph centric graph algorithms.

Gopher

BSP

Floe

GoFS

## Deployment Architecture

Gopher deployment consists of deploying three main components

* Resource Manager
* Coordinator
* Containers

Resource Manager responsible for resource allocation for processors. It will open TCP port 45001 for communication.

Coordinator responsible for workflow deployment and coordinating resource allocation tasks with the Resource Manager. Coordinator opens up port 45001 for communication.

Containers are the nodes that run the BSP processors. They should be co located with the GoFS data nodes. Containers should be able to communicate with each other via any open TCP ports.



Resource Manager





Containers/GoFS data nodes

Coordinator

Starting gopher includes starting Resource Manager , Coordinator and Containers. Then users must deploy Gopher Workflow in the system by sending the Gopher data flow graph to the coordinator.

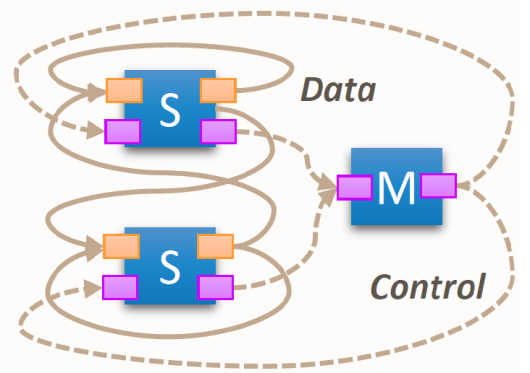
The coordinator will process the workflow graph and deploy BSP processors in the coordinator while coordinating with the Resource Manager.

## Gopher Data Flow

As mentioned in the previous section Gopher Is a BSP workflow deployed in Floe. In Floe Work flow tasks are called Pellets. BSP workflow consists of 3 types of Pellets

* BSPProcessorPellet – Act as BSP Processor. BPS processor takes care of executing sub-graph tasks for the sub-graphs in the graph partitions that is stored in its current machine. It will run multiple subgraph tasks in parallel based on the processing power and memory size if the container its running.
* ControlPellet - Act as coordinator between BSP processors to support barrier synchronization.
* FWDPellet - Entry point from the outside world to the workflow.

Following is a simple BSP workflow In Floe.



## API

Gopher provides a Subgraph centric BSP programming abstraction for users. Time series graphs are partitioned and distributed on several computers where each partition will have set of subgraphs. Gopher will operate on subgraphs in parallel. Communication across subgraphs are done via messaging. The user can send a message to a given sub-graph given by providing a vertex in that sub-graph with the  partition id of that subgraph.

The following section describes the current Gopher User APIs.

**Sub-Graph API**

Users will extend the subgraph api to implement the graph processing logic.

public interface IGopherSubGraph {

/\*\*

\* User implementation logic goes here.

\* To send message to a another partition use {@link #sentMessage(long, SubGraphMessage)}} see

\* {@link edu.usc.pgroup.floe.applications.gopher.api.SubGraphMessage} for more details about the message format.

\* To signal that current logic is done processing use {@link #voteToHalt()}

\* @param messageList List of SubGraphMessage which is intended for this sub graph.

\*/

public abstract void compute(List<SubGraphMessage> messageList);

/\*\*

\* Get the Current super step.

\* @return

\*/

public final int getSuperStep() ;

/\*\*

\* Signal that this subgraph finished processing. The system will come to a halt state once all the subgraphs come

\* to an halt state.

\*/

public final void voteToHalt();

/\*\*

\* Send Message to a given partition.

\* @param partitionId target partition

\* @param message data message for that partition

\*/

public final void sentMessage(long partitionId,SubGraphMessage message);

}

**Sub-Graph Message**

Subgraph message is used to communicate/coordinate among sub-graphs.

public interface SubGraphMessage<T> {

/\*\*

\* Add any application specific tag  
 \* @param tag application specific tag  
 \* @return Current Message Instance  
 \*/  
 public SubGraphMessage<T>    addTag(String tag);

/\*\*  
 \* Set the target vertex id this messages is referring to.  
 \* This vertex id will be used at the remote partition to determine the sub-graph that  
 \* this message will be dispatched to.  
 \* This message will be dispatched to the sub-graph which contain the target vertex  
 \* @param id target remote vertex id  
 \* @return Current Message instance.  
 \*/  
 public SubGraphMessage<T>    addTargetVertex(long id);

/\*\*  
 \* Get the Target Remote vertex id  
 \* @return target remote vertex id  
 \*/  
 public long getTargetVertex();

/\*\*  
 \* Get the application specific message tag.  
 \* @return message tag.  
 \*/  
 public String getTag();

/\*\*  
 \* Get the message data as a T  
 \* @return data as a T  
 \*/  
 public T[]    getData();

}

**Example**

Following code segment shows a simple example of calculating total vertex count in a large graph using Gopher.

Each subgraph calculates the number of vertices in the throat and send the count to the other subgraphs where upon arrival of the messages subgraph will update its current vertex count by adding that incoming count if it's not coming from the it. Each subgraph will write the calculated total to a file.

public void compute(List<SubGraphMessage> subGraphMessages) {  
 if(subGraphMessages == null || subGraphMessages.size() == 0) {

currentCount = subgraph.getTemplate().numVertices();

for(long part : partitions) {  
 String count = "" + partition.getId() + ":" + subgraph.getId() + ":" + currentCount;  
 SubGraphMessage<String> msg = new SubGraphMessage<String>(count.getBytes());  
 sentMessage(part,msg);  
 }

} else {

for(SubGraphMessage<String> msg : subGraphMessages) {

String count = new String(msg.getData());  
 String dataParts[] = count.split(":");

long partId = Long.parseLong(dataParts[0]);  
 long subId = Long.parseLong(dataParts[1]);

if(!(partition.getId() == partId && subgraph.getId() == subId)) {  
 currentCount += Integer.parseInt(dataParts[2]);  
 }

}

try {  
 File file = new File("vert-count.txt");  
 PrintWriter writer = new PrintWriter(file);  
 writer.write("Total Vertex Count :" + currentCount );  
 writer.flush();  
 writer.close();  
 } catch (FileNotFoundException e) {  
  logError(e);

}

voteToHalt();

}