Hama Streaming Protocol

Abstract

The Hama Streaming Protocol is a protocol which enables non-Java applications to write BSP functions just like in pure Java code. The communication takes place with plain strings and is therefore highly portable. Also this is the only difference to Hama Pipes, which is a native implementation for C/C++ fully based on low level binary communications and is therefore faster and has less overhead. In the following chapters, we will introduce the text based protocol and its bidirectional communication, also with the practical example of a Python implementation.
So we would be proud if other people take their time and implement the streaming protocol in other languages, for example in Scala or Ruby.

Streaming Basics

Hama Streaming is a special BSP implementation, which basically takes care of stream communication with the predefined protocol and by forking a child process with the given configuration. This configuration can be an interpreter, like python or bash, or a compiled binary. Hama now redirects the input and output streams of that child process to streams that are used for the protocol. Therefore you can't simply print by writing to STDOUT, but you can use the LOG function that redirects special log statements to the log of the Java task, this will be explained later.

The protocol is designed to mimic the Java API, for obvious reasons of documentation and convenience, so we suggest to make yourself comfortable with the Java API first and then get into coding the protocol. For additional information about the design and programming model, have a look at our Getting Started section in our wiki and the user documentation as a PDF.
http://wiki.apache.org/hama/GettingStarted#User_documentation_as_PDF
If you want to know how to run a user code within other environments, have a look in the python example at the end.

Text Based Protocol

The text based protocol is very easy. You have a finite amount of operations that are associated with a unique identifier (OP_CODE). Common operations in BSP are sending a message or starting the barrier synchronization. In Hama Streaming, every operation is ended with a newline ('\n') character, so everything is done with a single line of text. Of course there are operations that are going to use multiple lines to naturally separate information, however these are special cases.
In cases where the transferred information contains newline characters, you will run into problems, if you desperately need a fix for that, we want to add this in the next protocol version, if you can't wait you need to prepare the transferred data that it does not contain newlines.

Special cases, e.g. the initialization of the bsp phase are using a special text wrapper for the operation code "%" + OP_CODE + "%=", e.g. "%0%=" for the initial startup code. Note that this text wrapper is required for all communication from the child process to the BSP task, but not all BSP task communications use these special expressions.

Operation Codes

Here is a table of all op codes and their corresponding unique identifier, note that not all of them are in use, because the streaming protocol is working on top of the binary pipes protocol.

Operation Code Name	Operation Code identifier	Comment
START	0	First op code after fork
SET_BSP_JOB_CONF	1	Get configuration values
SET_INPUT_TYPES	2	Not used
RUN_SETUP	3	Start of the setup function
RUN_BSP	4	Start of the bsp function
RUN_CLEANUP	5	Start of the cleanup function
READ_KEYVALUE	6	Reads a key/value pair from input (Text Only)
WRITE_KEYVALUE	7	Writes a key/value pair to output (Text Only)
GET_MSG	8	Gets the next message in the queue
GET_MSG_COUNT	9	Get how many messages are in the queue
SEND_MSG	10	Send a message
SYNC	11	Start the barrier synchronization
GET_ALL_PEERNAME	12	Get all peer names
GET_PEERNAME	13	Get the peer name of the current peer
GET_PEER_INDEX	14	Get a peer name via index
GET_PEER_COUNT	15	Get how many peers are there
GET_SUPERSTEP_COUNT	16	Get the current Superstep counter
REOPEN_INPUT	17	Reopens the input to reread
CLEAR	18	Clears the messaging queue
CLOSE	19	Closes the protocol
ABORT	20	Not used
DONE	21	Closes the protocol if task is done
TASK_DONE	22	Yet another task is done op code
REGISTER_COUNTER	23	Not used (please create a new issue for that)
INCREMENT_COUNTER	24	Not used (please create a new issue for that)
LOG	25	Not implemented in Pipes, but in streaming it sends child logging to the Java task.

Acknowledgements

To detect whether the forked child has arrived at the next stage of an algorithm, we work with acknowledgements (short ACK), that have a special formatting.
ACKS are formally expressed as *"%ACK_" + OP_CODE + "%=",* where OP_CODE is the operation to acknowledge.

Initialization Sequence

After the child process has been forked from the Java BSP Task, a special initialization sequence is needed. Formally the sequence looks like this:

- START_OP_CODE from Java task
- Protocol number from Java task (currently actual version number is 0)
- SET_BSPJOB_CONF OP_CODE from Java task
- Number "N" of configuration items (note that this is a sum over number of keys and values)
- Now "N" lines follow, where the first line contains the configuration key, the following the value.
- Now the forked child can send an acknowledgement to the Java task

A sample communication could look like this // are comments and are not parts of the protocol:

```
%0%=                    // start op code from java task
0                       // protocol number
%1%=                    // set bsp configuration op code
2                       / number of configuration items to read
hama.bsp.child.opts     // sample key from configuration
-Xmx512m                // sample value from configuration
```

Now the forked child needs to acknowledge the start OP code by writing:

```
%ACK_0%=                // ACK'd start code
```

After the acknowledgement, we immediately start with the setup function.

Setup/BSP/Cleanup Sequence

A normal BSP task has three major steps:

- Setup (used to allocate resources, save configuration values as variables, etc.)
- BSP (the main logic of your algorithm)
- Cleanup (free the resources claimed during the whole task)

The protocol is the same for every of the three steps:

- X_OP_CODE from Java task // where X can be SETUP (3), BSP (4) or CLEANUP (5)
- Now the forked child can run the bsp task of the user algorithm
- The forked child can send an acknowledgement to the Java task that the step has finished

Of course here is the example with the setup function:

%3%= // setup op code from java task
// now call the users setup function, any communication can happen here
%ACK_3%= // ACK'd setup code

The three steps are called in sequential order, so after you have ACK'd the end of the setup, the Java code will immediately start telling you about that you need to start the bsp function. This applies also for the transition between bsp and cleanup function.

BSPPeer Functionality

People familiar with the Hama BSP API know that there is a context object which gives access to the BSP functionality like send or sync. We think this is a cool design and you should take care of implementing it as well, so you can pass this peer in all the user implemented functions.
Here is the table for the communication of the currently implemented BSP functionality:

BSP Primitive	Operation sequence	Response
Send	- %10%= - Destination peer name - Message as text	
Sync	- %11%=	After sync a special ACK will be in the next line: "%11%=_SUCCESS" So you should block until you received this.
getCurrentMessage	- %8%=	"%%-1%%" if no message was found, else the message as text
getSuperstepCount	- %16%=	The raw Superstep number
getMessageCount	- %9%=	The raw message count
getPeerName	- %13%= - Followed by an index (plain integer), -1 for the name of this peer.	The name of the peer as text
getPeerIndex	- %14%=	The raw index in the peer array
getAllPeerNames	- %12%=	A line how many peers are there (plain number), then n-lines with the peer names
getPeerCount	- %15%=	Plain number of how many peers are there
writeKeyValue	- %7%= - Key as line - Value as line	
readKeyValue	- %6%=	Key and Value in two separate following lines. If no input available, both are equal to "%%-1%%".
reopenInput	- %17%=	

Closing Sequences

To determine if the child process successfully finished its computations, we have a closing sequence that is expected after you acknowledged your cleanup.
The closing sequence is basically just sending TASK_DONE and DONE command to the Java process. After receiving this, the Java process will do the normal cleanup and finish the task itself, after sending these codes you can gracefully shutdown your process by letting it exit with a zero status.

The end sequence looks like this:

```
%22%=          // task done
%21%=          // completely done
```

Congratulation, you are now able to implement Hama streaming in other languages.

Appendix

Known implementations

Hama Streaming for Python: https://github.com/thomasjungblut/HamaStreaming

Running user code in the Python environment

In the Python runtime, you can pass a .py filename to the BSPRunner as argument, using __import__ you can import the class, get the class via getattr and get an instance by instantiating this class.

Looks basically like that:

```
className = sys.argv[1]
module = __import__(className)
class_ = getattr(module, className)
bspInstance = class_()
```

Now you can pass this instance to the handler that calls back the user functions in the python implementation this is done by the peer itself.