PHISH Documentation

http://www.sandia.gov/~sjplimp/phish.html

Table of Contents

Table of Contents

Table of Contents

PHISH Tales

(documentation for the PHISH library)

Version info:

The PHISH "version" is the date when it was released, such as 1 Nov 2012. PHISH is updated continuously. Whenever we fix a bug or add a feature, we release it immediately, and post a notice on this page of the WWW site. Each dated copy of PHISH contains all the features and bug-fixes up to and including that version date. Each time you use the bait.py tool, the version date is printed to the screen. It is also in the file bait/version.py and in the PHISH directory name created when you unpack a tarball.

- If you browse the HTML or PDF doc pages on the PHISH WWW site, they always describe the most current version of PHISH.
- If you browse the HTML or PDF doc pages included in your tarball, they describe the version you have.

PHISH stands for Parallel Harness for Informatic Stream Hashing. The phishy metaphor is meant to evoke the image of many small minnows (programs) swimming in a stream (of data).

PHISH is a lightweight framework which a set of independent processes can use to exchange data as they run on the same desktop machine, on processors of a parallel machine, or on different machines across a network. This enables them to work in a coordinated parallel fashion to perform computations on either streaming, archived, or self-generated data.

The PHISH distribution includes a simple, portable library for performing data exchanges in useful patterns either via MPI message-passing or ZMQ sockets. PHISH input scripts are used to describe a data-processing algorithm, and additional tools provided in the PHISH distribution convert the script into a form that can be launched as a parallel job.

PHISH was developed at Sandia National Laboratories, a US Department of Energy facility, with funding from the DOE. It is an open-source code, distributed freely under the terms of the Berkeley Software Distribution (BSD) License.

The authors of PHISH are Steve Plimpton and Tim Shead who can be contacted at sjplimp, tshead at sandia.gov. The PHISH WWW Site at http://www.sandia.gov/~sjplimp/phish.html has more information about the code and its uses.

The PHISH documentation is organized into the following sections. If you find errors or omissions in this manual or have suggestions for useful information to add, please send an email to the developers so we can improve the PHISH documentation.

PDF file of the entire manual, generated by htmldoc

1. Introduction

This section explains what the PHISH software package is and why we created it. It outlines the steps to creating your own PHISH program, and gives a simple example of using PHISH to perform a parallel calculation. These are the topics discussed:

1.1 Motivation

Informatics is data-driven computing and is becoming more prevalent, even on large-scale parallel machines traditionally used to run scientific simulations. It can involve processing large archives of stored data or data that arrives on-the-fly in real time. The latter is often referred to as "streaming" data. Common attributes of streaming data are that it arrives continuously in a never-ending stream, its fast incoming rate requires it be processed as it arrives which may limit the computational effort per datum that can be expended, and its high volume means it cannot be stored permanently so that individual datums are examined and discarded.

A powerful paradigm for processing streaming data is to use a collection of programs, running as independent processes, connected together in a specified communication topology. Each process receives datums continuously, either from the stream itself, or read from a file, or sent to it from other processes. It performs calculations on each datum and may choose to store "state" internally about the stream it has seen thus far. It can send the datum on to one or more other processes, either as-is or in an altered form.

In this model, a data-processing algorithm can be expressed by choosing a set of processes (programs) and connecting them together in an appropriate fashion. If written flexibly. individual programs can be re-used in different algorithms.

PHISH is a small software package to make the task of designing and developing such algorithms easier, and allowing the resulting program to be run in parallel, either on distributed memory platforms that support MPI message passing, or on a collection of computers that support socket connections between them.

PHISH stands for Parallel Harness for Informatic Stream Hashing.

Here is what these words mean, in the PHISH context. "Parallelism" can be achieved by using multiple copies of processes, each working on a part of the stream, or by using the memory of multiple processes to store state about the stream of data. It is a framework or "harness" for connecting processes in a variety of simple, yet powerful, ways that enable parallel data processing. While it is designed with "streaming" "informatics" data in mind, it can also be used to process archived data from files or in a general sense to perform a computation in stages, using internally generated data of any type or size. "Hashing" refers to sending datums to specific target processes based on the result of a hash operation, which is one means of achieving parallelism.

It is important to note that PHISH does not replace or even automate the task of writing code for the individual programs needed to process data, or of designing an appropriate parallel algorithm to perform a desired computation. It is simply a library that processes can call to exchange datums with other processes, and a setup tool that converts an input script into a runnable program that can be easily launched in parallel.

Our goal in developing PHISH was to make it easier to process data, particularly streaming data, in parallel, on distributed-memory or geographically-distributed platforms. And to provide a framework to quickly experiment with parallel informatics algorithms, either for streaming or archived data. Our own interest is in graph algorithms but various kinds of statistical, data mining, machine learning, and anomaly detection algorithms can be formulated for streaming data, in the context of the model described above. We hope PHISH can be a useful tool in those settings as well.

1.2 PHISH lingo

The name PHISH was also chosen because it evokes the image of fish (programs) swimming in a stream (of data). This unavoidably gives rise to the following PHISH lingo, which we use without apology throughout the rest of the documentation:

- minnow = a (typically small) stand-alone application, run as a process
- school = a set of duplicate minnows, swimming (working) together in coordinated fashion
- hook = a connection between two schools in a defined communication pattern
- net(work) = a PHISH program, consisting of one or more schools, hooked together to perform a calculation
- bait.py = a tool for creating PHISH nets from an input script
- wrapper = a wrapper of the PHISH library for Python
- tales = the PHISH manual

1.3 PHISH pheatures

The model described above is not unique to PHISH. Many programs provide a framework for enabling data to flow between computational tasks interconnected by "pipes" in a dataflow kind of paradigm. Visualization programs often use this model to process data and provide a GUI framework for building a processing pipeline by connecting the outputs of each computational node to the inputs of others. The open source Titan package, built on top of VTK, is one example, which provides a rich suite of computation methods, both for visualization and data processing. The commercial InfoSpheres tool from IBM uses a similar dataflow model, and is designed for processing streaming data at high rates. Twitter recently released an open-source package called "Twitter Storm" which has been advertised as Hadoop for streaming data, since it enables streaming MapReduce-style computations and runs with fault-tolerance on top of a parallel file system like HDFS. PHISH has many conceptual similarities to Storm, though PHISH has fewer features, such as no support for fault tolerance.

Dataflow frameworks like these are often designed to run as a single process or in parallel on a shared memory machine. The computational nodes in the processing pipeline are functions called as needed by a master process, or launched as threads running in parallel.

By contrast, PHISH minnows (computational nodes in the processing pipeline), are independent processes and the PHISH library moves data between them via "messages" which requires copying the data, either using the MPI message-passing library or sockets. This allows PHISH programs to be run on a broader range of hardware, notably distributed-memory parallel platforms, but also incurs a higher overhead for moving data from process to process.

The following list highlights additional PHISH pheatures:

* The PHISH package is open-source software, distributed under the Berkeley Software Development (BSD) license. This effectively means that anyone can use the software for any purpose, including commercial redistribution.
* The PHISH library is a small piece of code (few 1000 lines), with a compact API (couple dozen functions). It has a C interface, so that it can be easily called from programs written in a variety of languages (C, C++, Fortran, Python, etc). The library can be compiled on any platform with a C++ compiler.
* The PHISH library comes in two flavors with the same API: one based on message passing via the MPI library, the other based on sockets via the open-source ZMQ library. This means you need one or both of these packages (MPI, ZMQ) installed on your machine to build a minnow (process) that uses the PHISH library.
* PHISH nets (programs) consist of one or more collections of minnows (schools), hooked together in defined communication patterns, to encode an algorithm or computation, The toplogy of a PHISH net is specified in an input script, which is text files with a simple command syntax.
* PHISH minnows can define one or more input and output ports for sending and receiving datums. This allows schools of minnows to be hooked together in a variety of communication patterns.
* PHISH minnows can be written to operate on datums of various types (e.g. integers or floating-point values or strings). This allows minnows to be re-used in various PHISH nets.
* A PHISH wrapper (pun intended) for Python is provided, so that minnows that call the PHISH library can be written in Python. Minnows written in different languages (e.g. C++ or Python) can be used interchangeably in a PHISH input script.
* The PHISH library exchanges data between minnows with strict data typing rules, so that minnows can be written in different languages (e.g. C++ vs Fortran vs Python) or run on different machines (4-byte vs 8-byte integers).
* PHISH input scripts are processed via a tool called bait.py tool which can also run them as an MPI or socket-based program.
* PHISH input scripts use a hook command which allows data to be exchanged in various patterns between schools of minnows. This enables parallelism in data processing to be easily expressed and exploited.
* PHISH nets can be run on a single processor, so long as the OS supports multiple processes. They can be run on a multicore box. They can be run on any distributed-memory or shared-memory platform that supports MPI or sockets. Or they can be run on a geographically dispersed set of machines that support socket connections.
* A PHISH net can look for incoming data on a socket port. It can likewise export data to a socket port. This means that two or more PHISH nets can be launched independently and exchange data. This is a mechanism for adding/deleting minnows (processes) to/from a calculation on the fly.
* PHISH minnows are included that wrap non-PHISH applications that read from stdin and/or write to stdout. This allows such an application to be used in a PHISH net and exchange data with other minnows.

1.4 Steps to create and run a PHISH net

The PHISH package contains a library and a tool for defining and running PHISH nets. These are the steps typically used to perform a calculation, assuming you have designed an algorithm that can be encoded as a series of computational tasks, interconnected by moving data between them.

1. Build the PHISH library
2. Write and build one or more minnows that call the PHISH library.
3. Write an input script that defines a PHISH net as minnows, schools and communication patterns between them.
4. Use the bait.py tool to process and run the input script

Step (1): An overview of the PHISH library and instructions for building it are given in this section.

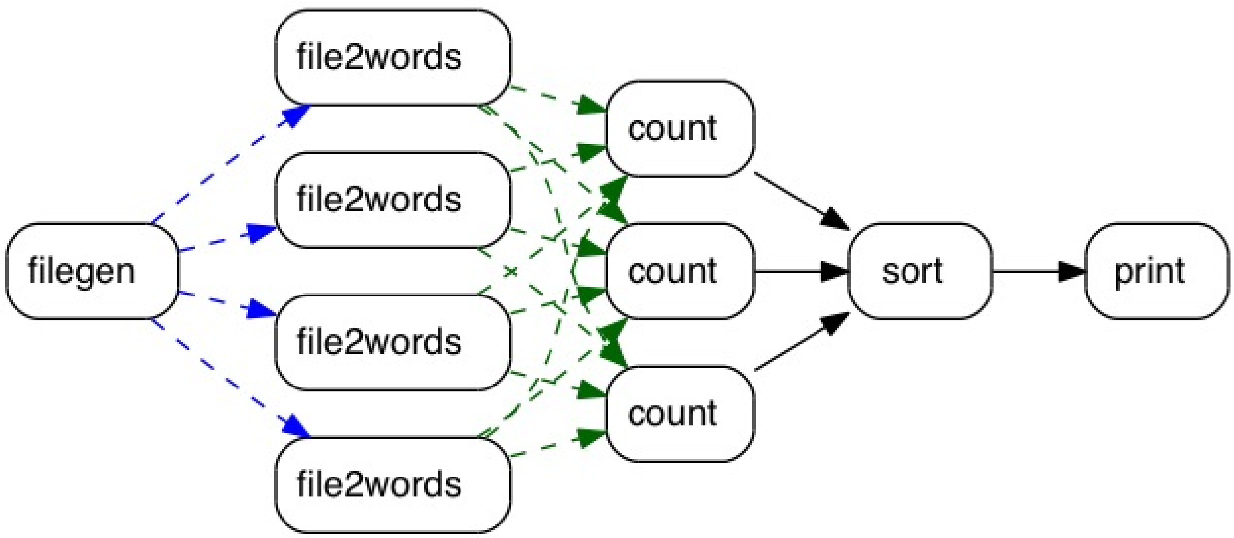
Step (2): A minnow is a stand-alone program which makes calls to the PHISH library. An overview of minnows, their code structure, and how to build them, is given in this section. The API for the PHISH library is given in this section, with links to a doc page for each function in the library.

Step (3): The syntax and meaning of commands used in PHISH input scripts are described in this section.

Step (4): The bait.py tool, its command-line options, and instructions on how to use it, are described in this section. Before using it the first time, one or more backend libraries must be built, which are in the src directory. This can be done as part of step (1).

1.5 Simple example

The steps outlined in the preceding section are somewhat abstract. Here is a concrete example of using a PHISH program to count the number of times different words appear in a corpus of text files. This is effectively a MapReduce operation, where individual minnow processes perform the map() and reduce() functions. This is a diagram of how 5 different kinds of minnows can be connected together to perform the computation:



Code for all 5 of these minnows is in the example directory of the PHISH distribution, both in C++ and Python. The *filegen* minnow takes a list of files and/or directories as user input, searches them recursively, and generates a series of filenames. The filenames are sent one-at-a-time to one of several *file2words* minnows. Each receives a filename as input, opens and reads the content, and parses it into words. Each word is hashed and sent to a specific *count* minnow, as indicated by the all-to-all green arrows. The key point is that each *count* minnow will receive all occurrences of a subset of possible words. It uses an internal hash table to count the occurrences of each word it receives. Note that parallelism is enabled by invoking multiple copies of the *file2words* and *count* minnows.

When the *filegen* minnow sends the last filename, it sends a "done" message to each of the *file2words* minnows. When they receive a "done" message, they in turn send a "done" message to each *count* minnow. When a *count* minnow has received a "done" message from all the *file2words* minnows, it sends its entire list of unique words and associated counts to the *sort* minnow, followed by a "done" message. When the *sort* minnow has received "done" message from all the upstream *count* minnows, it knows it has received all the unique words in the corpus of documents, and the count for each one. It sorts the list by count and sends the top *N* to the *print* minnow, followed by a "done" message. *N* is a user-defined parameter. The *print* minnow echoes each datum it receives to the screen or a file, until if receives a "done" message. At this point all minnows in the school have been shut down.

More details about this example are discussed in subsequent sections of the manual:

- In this section of the bait.py tool doc page, the PHISH input script that encodes the minnows and communication connections of the above diagram is discussed, and its processing by the bait.py tool.
- In this section of the PHISH Minnows doc page, the code for the *count* minnow is discussed in detail, to illustrate what calls it makes to the PHISH library to send and receive datums.
- In this section of the PHISH Library doc page, the format of datums exchanged between minnows is discussed.

Note that like a MapReduce, the PHISH program runs in parallel, since there can be N *file2words* minnows and M *count* minnows where N >= 1, M >= 1, and N = M is not required. This is similar to the option in Hadoop to vary the numbers of mappers and reducers.

However, there are also some differences between how this PHISH program works as compared to a traditional MapReduce, e.g. as typically performed via Hadoop or the MapReduce-MPI library.

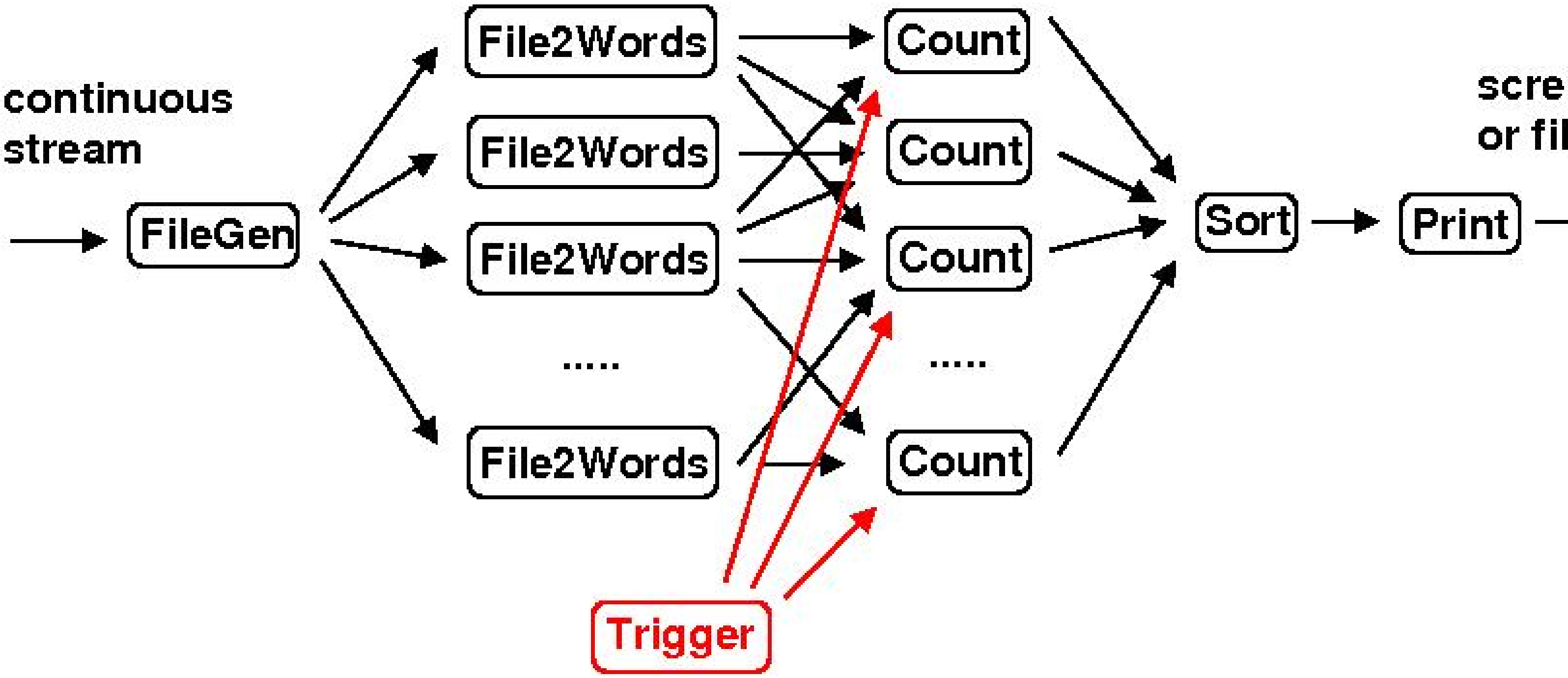
In a traditional MapReduce, the "map" stage (performed by the *file2words* minnows) creates a huge list of all the words, including duplicates, found in the corpus of documents, which is stored internally (in memory or on disk) until the "mapper" process is finished with all the files it processes. Each mapper then sends chunks of the list to each "reduce" process (performed by the *count* minnows). This is the "shuffle" phase of a Hadoop MapReduce. The reducer performs a merge sort of all the words in the chunks it receives (one from each mapper). It can then calculate the count for each unique word.

In contrast, the PHISH program operates in a more fine-grainded fashion, streaming the data (words in this case) through the minnows, without ever storing the full data set. Only a small list of unique words is stored (by the *count* minnows), each with a running counter. PHISH exchanges data between minnows via many tiny messages (one word per message), whereas a traditional MapReduce would aggregate the data into a few large messages.

This is a simplistic explanation; a fuller description is more complex. Hadoop, for example, can operate in streaming mode for some forms of MapReduce operations, which include this wordcount example (MapReduce operations where the "reducer" needs all data associated with a key at one time, are not typically amenable to a streaming mode of operation.) The PHISH minnows used in this school could be modified so as to aggregate data into larger and fewer messages. Likewise, in a traditional MapReduce, large intermediate data sets can be stored out-of-core. PHISH does have the capability to do that unless a minnow is written that writes information to disk and retrieves it.

However the fundamental attributes of the PHISH program are important to understand. Data moves continuously, in small chunks, through a school of minnows. Each minnow may store "state" information about the data it has previously seen, but typically not all the data itself. "State" is typically limited to information that can be stored in-memory, not on disk. This is because for streaming data, too much data arrives too quickly, for a minnow to perform much computation before discarding it or sending it on to another minnow.

Here is a diagram of a variant of the wordcount operation that illustrates how PHISH can be used to process continuous, streaming data. The PHISH program in this case might run for days or weeks, without using the "done" messages described above.



In this case the *filegen* minnow is continuously seeing new files appear in directories it monitors. The words in those files are processed as they appear. A *Trigger* minnow has been added which accepts user queries, e.g. via a keyboard or a socket connection. When the user makes a request (hits a key), a message is sent to each of the *count* minnows on a different input port than it receives words from the *file2words* minnows; see this section of the PHISH Minnows doc page for a discussion of ports. The message triggers the *count* minnows to send their current unique word/count list to the *sort* minnow which is sorted and printed via the *print* minnow.

The PHISH job now runs continuously and a user can query the current top N words as often as desired. The *filegen*, *count*, and *sort* minnows would have to be modified, but only in small ways, to work in this mode. Additional logic could be added (e.g. another user request) to re-initialize counts or accumulate counts in a time-windowed fashion.

1.6 Acknowledgments and citations

PHISH development has been funded by the US Department of Energy (DOE), through its LDRD program at Sandia National Laboratories.

The following paper describes the basic ideas in PHISH. If you use PHISH in your published work, please cite this paper and include a pointer to the PHISH WWW Site (http://www.sandia.gov/~sjplimp/phish.html):

S. J. Plimpton and T. Shead, "Streaming data analytics via message passing with application to graph algorithms", submitted to J Parallel and Distributed Compuing, 2012.

PHISH was developed by the following individuals at Sandia:

- Steve Plimpton, sjplimp at sandia.gov
- Tim Shead, tshead at sandia.gov

Source code for PHISH is freely available for download from the PHISH web site and is licensed under the modified Berkeley Software Distribution (BSD) License. This basically means it can be used by anyone for any purpose. See the LICENSE file provided with the distribution for more details.

2. Bait.py Tool

Bait.py is a Python program which parses a PHISH input script and uses a dynamically-loaded backend to directly run a PHISH net and perform a calculation, or create a script that can be used to do the same. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows via its input and output ports. A "net" is collection of schools of minnows.

There are Bait backends for running a PHISH net using MPI, running a PHISH net using ZMQ, generating configuraiton files for MPI or ZMQ, and generating a dotfile that can be converted into a diagram of a PHISH net via the GraphViz tool.

You can edit the input script or pass it different parameters via bait.py command-line arguments to change the calculation. Re-running bait.py will run a new net or create a new script.

The remainder of this page discusses how bait.py is used and how a PHISH input script is formatted. The input script commands recognized by bait.py have their own doc pages.

2.1 Input script commands

These are the input script commands recognized by bait.py:

- variable
- set
- minnow
- hook
- school

2.2 Building and running bait.py

Before using bait.py for the first time, one or more backend libraries must be built which bait.py uses for interfacing to MPI and/or ZMQ. This creates shared libraries which your Python must also be able to find.

The easiest way to build all of PHISH, including the bait backend libraries, is to use the cross-platform CMake build system. We recommend building PHISH with a separate build directory:

```
$ tar xzvf phish.tar.gz -C ~/src
$ mkdir ~/build/phish
$ cd ~/build/phish
$ ccmake ~/src/phish-14sep12
```

Then, in the CMake curses interface, configure the build, generate makefiles, and build phish:

```
$ make
```

You can also build one or more of the backend libraries from the src directory of the distribution by typing one or more of these lines:

```
make -f Makefile.machine baitmpi
make -f Makefile.machine baitmpiconfig
make -f Makefile.machine baitzmq
make -f Makefile.machine baitgraph
make -f Makefile.machine baitnull
```

where "machine" is the name of one of the Makefiles in the directory. These should produce files like libphish-bait-mpi.so or libphish-bait-zmq.so. See the discussion of the --backend command-line switch in the next section, for the difference between the various backend options. See the discussion in this section if none of the provided Makefiles are a match to your machine.

When you run bait.py, your Python must be able to find the appropriate backend shared library. The simplest way to do this is to add a line to your shell start-up script.

For csh or tcsh, add a line like this to your .cshrc file:

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/sjplimp/phish/src
```

For bash, add a line like this to your .bashrc file:

```
export LD_LIBRARY_PATH $LD_LIBRAY_PATH:/home/tshead/build/phish/src
```

For OSX systems, use DYLD_LIBRARY_PATH instead of LD_LIBRARY_PATH.

After editing your shell start-up script, be sure to invoke it, e.g. source .cshrc.

See the discussion in this section for an alternative way to do this.

You are now ready to use the bait.py tool. It is a Python script in the bait directory of the PHISH distribution. Like any Python script you can run it in one of two ways:

```
bait.py --switch value(s) ... <in.script
python bait.py --switch values ...  <in.script
```

For the first case, you need to insure that the first line of bait.py gives the correct path to the Python installed on your machine, e.g.

```
#!/usr/local/bin/python
```

and that the bait.py file is executable, e.g.

```
chmod +x bait.py
```

Normally you will want to invoke bait.py from the directory where your PHISH input script is, so you may need to prepend bait.py with a path or make an alias for running it conveniently.

The switch/value command-line arguments recognized by bait.py are discussed in the next section.

2.3 Command-line arguments

These are the command-line arguments recognized by bait.py. Each is specified as "-switch value(s)". Each switch has an abbreviated form; several of them have default settings.

- -h or --help
- -b BACK or --backend BACK
- -l LAUNCHER or --launch LAUNCHER
- -p PATH1:PATH2:... or --path PATH1:PATH2:...
- -s NAME VALUE or --set NAME VALUE
- -x SUFFIX or --suffix SUFFIX
- -v NAME VALUE or --variable NAME VALUE
- --verbose

Use *--help* to display a help message and exit.

Use *--backend* to select the desired bait.py backend. The choice of backend defines how the input script will be interpreted to run a PHISH net. Current choices for BACK are "graphviz", "mpi", "mpi-config", "null", and "zmq". We plan to add a "zmq-config" option.

- The graphviz backend will write a file in DOT format to stdout. You can process this file using any of the GraphViz tools to create a diagram of your PHISH net, useful for documentation or presentations.
- The mpi backend will run your PHISH net immediately using the mpiexec command, which must be available somewhere on your system PATH.
- The mpi-config backend will write an mpiexec compatible config file to stdout. You can then run your PHISH net any time by passing the generated file to mpiexec.
- The null backend is a do-nothing backend that is useful for troubleshooting. For example, you can combine the --verbose option with the null backend to confirm that variables are expanded correctly in your PHISH input script.
- The zmq backend will run your PHISH net immediately using ZMQ sockets. Note that there is a variable called "hostnames" that must be set to use the ZMQ backend; see the variable doc page for details.

The *--launch* option will use the program LAUNCHER to invoke all the minnows. This is useful if the minnow is a Python script, in which case --launch python will launch the minnow using Python. LAUNCH can be multiple words if desired, e.g. --launch python -x.

The *--path* option specifies a colon-separated list of one or more directories as PATH1, PATH2, etc. When bait.py processes each minnow, as specified by the minnow command, it looks for the minnow's executable file in this list of directories, so that it can write it to the launch script with an absolute path name.

Use *--set* to set an option for the input script that is the same as if the set command had been used in the input script with NAME and VALUE. For example, --set memory 5 is the same as using "set memory 5" in the input script to specify the maximum datum size to 5 Kbytes. A value specified in the input script will override a command-line setting.

Use *--suffix* to supply a SUFFIX string that will be appended to the name of each minnow executable in your input script. This is useful when you have minnow executables that have been built using different communication backends - for example, if you have a minnow "foo.c", you might link it against the MPI and ZMQ backends to produce two executables, "foo-mpi", and "foo-zmq". Using the *--suffix* option, you can create a single PHISH input script and run it against either executable by specifying --suffix=-mpi or --suffix=-zmq. It is also useful if a minnow is a Python script, ending in ".py", in which case you could specify --suffix=.py and use the *--launch* option to invoke the minnow with Python.

The *--variable* switch defines a variable that can be used within the script. It can be used multiple times to define different variables with NAME and VALUE. A variable command can also be used in the input script itself. The VALUE specified on the command-line will override the value of a variable with the same NAME in the input script, which allows you to set a default value in the input script and overrided it via the command line.

The variable NAME and VALUE are any alphanumeric string. A list of strings can also be assigned to it, e.g. a series of filenames. For example,

```
bait.py --variable files *.cpp <in.phish
```

creates the variable named "files" containing a list of all CPP files in the current directory.

Note that there is a variable called "hostnames" that must be set to use the ZMQ backend; see the variable doc page for details.

The *--verbose* option causes bait.py to produce verbose output while processing your input script. The verbose output will vary depending on the backend in use, and will be written to stderr.

2.4 Input script syntax and parsing

A PHISH input script is a text file that contains commands, typically one per line.

Blank lines are ignored. Any text following a "#" character is treated as a comment and removed, including the "#" character. If the last printable character in the line is "&", then it is treated as a continuation character, the next line is appended, and the same procedure for stripping a "#" comment and checking for a trailing "&" is repeated.

The resulting command line is then searched for variable references. A variable with a single-character name, such as "n", can be referenced as $n. A variable with a multi-character name (or single-character name), such as "foo", is referenced as ${foo}. Each variable found in the command line is replaced with the variable's contents, which is a list of strings, separated by whitespace. Thus a variable "files" defined either by a bait.py command-line argument or the variable command as

```
-v files f1.txt f2.txt f3.txt
variable files f1.txt f2.txt f3.txt
```

would be substituted for in this command:

```
minnow 1 filegen ${files}
```

so that the command becomes:

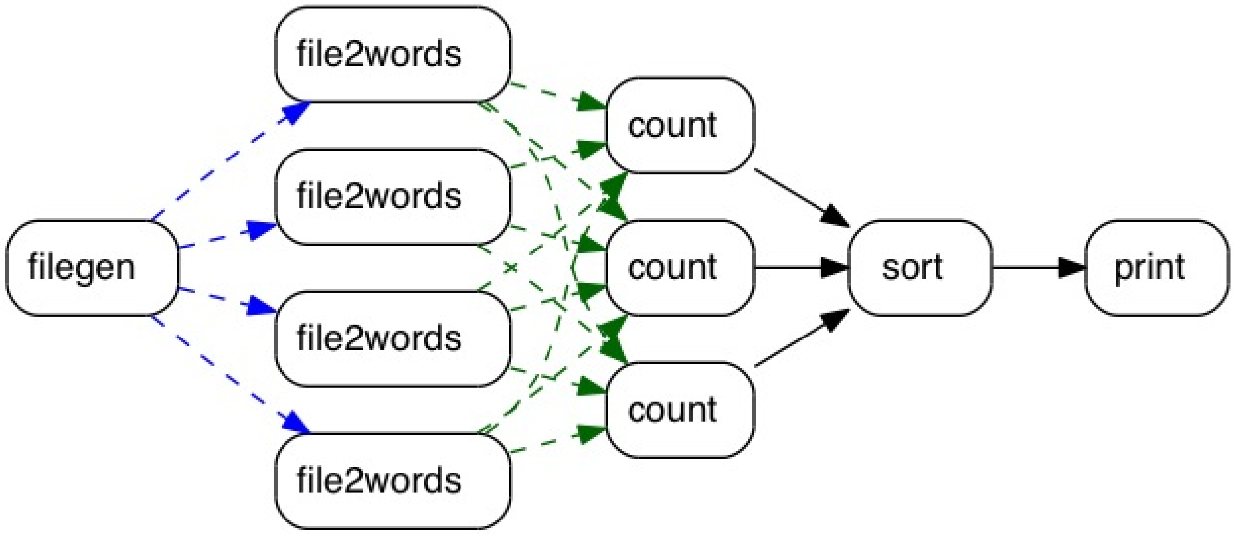
```
minnow 1 filegen f1.txt f2.txt f3.txt
```

After variable substitution, a single command is a series of "words" separated by whitespace. The first word is the command name; the remaining words are arguments. The command names recognized by bait.py are listed above. Each command has its own syntax; see its doc page for details.

With one exception, commands in a PHISH input script can be listed in any order. The script is processed by bait.py after the entire script is read. The exception is that a variable cannot be used before it is defined.

2.5 Simple example

This section of the Introduction doc page, discussed this diagram of a PHISH calculation for counting the number of times words appear in a corpus of files, performed as a streaming MapReduce operation:



This is the PHISH input script in example/in.wordcount that represents the diagram:

```
# word count from files
# provide list of files or dirs as -v files command-line arg

minnow 1 filegen ${files}
minnow 2 file2words
minnow 3 count
minnow 4 sort 10
minnow 5 print

hook 1 roundrobin 2
hook 2 hashed 3
hook 3 single 4
hook 4 single 5

school 1 1
school 2 5
school 3 3
school 4 1
school 5 1
```

The minnow commands list the 5 different minnows used. Note the use of the ${files} variable to pass a list of filenames or directories to the *filegen* minnow.

The hook commands specify the communication pattern used bewteen different schools of minnows. The key pattern for this example is the *hashed* style, which allows the *file2words* minnow to pass a "key" (a word) to the PHISH library. The library hashes the word to determine which *count* minnow to send the datum to.

The school commands specify how many instances of each minnow to launch. Any number of *file2words* and *count* minnows could be specified.

When this script is run thru bait.py in the example directory, as

```
../bait/bait.py --backend mpi-config -v files in.* -p ../minnow <in.wc > outfile
```

using -mpi-config as the backend, then bait.py produces the following lines in outfile. (Note that if --backend mpi is used, bait.py will launch the parallel job immediately after processing it.)

```
-n 1 ../minnow/filegen in.bottle in.cc in.cc.jon in.filelist in.pp in.rmat in.slow in.wc in.wrapsink
-n 5 ../minnow/file2words --phish-backend mpi --phish-minnow file2words 2 5 1 --phish-in 1 0 0 round
-n 3 ../minnow/count --phish-backend mpi --phish-minnow count 3 3 6 --phish-in 5 1 0 hashed 3 6 0 --
-n 1 ../minnow/sort 10 --phish-backend mpi --phish-minnow sort 4 1 9 --phish-in 3 6 0 single 1 9 0 -
-n 1 ../minnow/print --phish-backend mpi --phish-minnow print 5 1 10 --phish-in 1 9 0 single 1 10 0
```

which is the format of an mpiexec config file. There is one line per minnow, as defined by the input script. The "-n N" specifies how many copies of the minnow will be invoked. The next argument is the name of the minnow executable, followed by any minnow arguments, followed by backend-specific arguments such as "-minnow", "-in", and "-out" that encode the communication patterns between the minnows.

This outfile can be launched via the mpiexec command as:

```
mpiexec -configfile outfile
```

for MPICH, or as

```
mpiexec `cat outfile`
```

for OpenMPI. (Note that if --backend mpi is used, bait.py will launch the parallel job immediately after processing it.)

This will launch 11 independent processes as an MPI job. Each process will call the PHISH library to exchange datums with other processes in the pattern indicated in the diagram. The datum exchanges will be performed via MPI\Send() and MPI_Recv() calls since the MPI backend of the PHISH library is being invoked.

3. PHISH Minnows

In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library. Minnows are typically small programs which perform a single task, e.g. they parse a string searcing for keywords and store statistics about those keywords. But they can also be large programs which perform sophisticated computations and make only occasional calls to the PHISH library. In which case they should probably be called "sharks" or "whales" ...

An individual minnow is part of a "school" of one or more duplicate minnows. One or more schools form a PHISH "net(work)" which compute in a coordinated fashion to perform a calculation. Minnows communicate with each other to exchange data via calls to the PHISH library.

This doc page covers the following topics:

3.1 List of minnows

This is a list of minnows in the minnow directory of the PHISH distribution. Each has its own doc page. Some are written in C++ (*.cpp), some in Python (*.py), some in both. If provided in both languages, their operation is identical, with any exceptions noted in the minnow doc page:

- count
- file2words
- filegen
- print
- slowdown
- sort

These are also 3 special minnows which can wrap stand-alone non-PHISH applications which read from stdin and write to stdout, so that they can be used as minnows in a PHISH net and communicate with other minnows:

- wrapsink
- wrapsource
- wrapss

These are also two simple codes which can be compiled into stand-alone non-PHISH executables. They are provided as examples of applications that can be wrapped by the "wrap" minnows:

- echo
- reverse

3.2 Code structure of a minnow

The easiest way to understand how a minnow works with the PHISH library, is to examine a few simple files in the minnow directory. Here we list the count.py minnow, which is written in Python. There is a also a count.cpp minnow, written in C++, which does the same thing. The purpose of this minnow is to count occurrences of strings that it receives as datums:

```
1    #!/usr/local/bin/python as path to Python if desired
2
3    import sys
4    import phish
5
6    def count(nvalues):
7      if nvalues != 1: phish.error("Count processes one-value datums")
8      type,str,tmp = phish.unpack()
9      if type != phish.STRING:
10       phish.error("Count processes string values")
11     if hash.has_key(str): hashstr = hashstr + 1
12     else: hashstr = 1
13
14   def dump():
16     for key,value in hash.items():
17       phish.pack_int(value)
18       phish.pack_string(key)
19       phish.send(0)
20
21   args = phish.init(sys.argv)
22   phish.input(0,count,dump,1)
23   phish.output(0)
24   phish.check()
25
26   if len(args) != 0: phish.error("Count syntax: count")
27
28   hash =
29
30   phish.loop()
31   phish.exit()
```

On line 4, the Python minnow imports the phish module, which is provided with the PHISH distribution. Instructions on how to use this module, which wraps the C-interface to the PHISH library, are given in this section of the documentation.

The main program begins on line 21. The call to the phish.init is typically the first line of a PHISH minnow. When the minnow is launched, extra PHISH library command-line arguments are added which describe how the minnow will communicate with other minnows. These are stripped off by the phish.init function, and the remaining minnow-specific arguments are returned as "args". The phish.input and phish.output functions setup the input and output ports used by the minnow. A port is a communication channel by which datums arrive from other minnows or can be sent to other minnows. The PHISH input script sets up these connections, but from the minnow's perspective, it simply receives datums on its input port(s) and writes datums to its output port(s). See the next section for more discussion of ports.

There should be one call to phish.input for each input port the minnow uses. And one call to phish.output for each output port it uses. The call to the phish.check function on line 24 insures that the minnow as written is compatible with the way it is used in the PHISH input script, i.e. that the necessary input and output ports have been defined with valid hook styles.

The phish.input call specifies a callback function that the PHISH library will invoke when a datum arrives on that input port. In this case, the count minnow defines a count() callback function which stores a received string in a hash table (Python dictionary) with an associated count of the number of times it has been received.

On line 28, an empty hash table is initialized, and then the phish.loop function is called. This gives control to the PHISH library, which will wait for datums to be received, invoking the appropriate callback function each time one arrives.

The call to phish.input also defines a callback to the dump() function which is invoked when input port 0 is closed. This occurs when upstream minnows send the requisite number of "done" messages to the port. The dump() function sends the contents of the hash table to output port 0, one datum at a time. Each datum contains a unique string and its count.

The phish.loop function returns after invoking dump() and when all input ports are closed. The count minnow then calls the phish.exit function which will close its output port(s), and send "done" messages to downstream minnows connected to those ports.

This code structure is typical of many minnows:

- A beginnning section with a call to phish.init, definitions of input/output ports, and a call to phish.check. Then a call to phish.loop or phish.probe or phish.recv to receive datums. This is unnecessary if the minnow only generates datums, i.e. it is a source of data, but not a consumer of data.
- One or more callback functions unpack datums via the phish.unpack function, process their content, store state, and send messsages via phish.pack and phish.send functions.
- After phish.loop exits, the minnow shuts down via a call to phish.close or phish.exit and terminates. See this section for more discussion of shut down procedures.

3.3 Communication via ports

As discussed above, ports are input/output communication channels by which a minnow receives datums from an upstream minnow or sends datums to a downstream minnow.

The number of ports that can be configured by a minnow varies between PHISH library backends. The ZMQ version of the library supports an unlimited number of input and output ports, while the MPI version of the library supports up to MAXPORT number of input ports and MAXPORT number of output ports. MAXPORT is a hardwired value in src/phish-mpi.cpp which is set to 16. It can be changed if needed, but note that all minnows which use the MPI version of the library must be re-built since they must all use a consistent value of MAXPORT when run together in a PHISH net.

Note that a PHISH input script may connect a particular minnow to other minnows in a variety of ways. This applies to both the styles of hooks that are specified and the number of minnows on the other end of each hook. Thus it is possible for the user to specify hooks in the input script which the minnow does not support or even define. Similarly, the input script may cause other minnows to send datums to the minnow which it does not expect or is unable to interpret. This means a minnow should be coded to follow these rules:

- It should define each input port it receives datums on as "required" or "optional", via the phish.input function. This will generate erros if the PHISH input script is incompatible with the minnow.
- It should define each output port it sends datums to, via the phish.output function. This will also generate errors for incompatible PHISH input scripts, though the use of output ports by a script is always optional.
- The minnow should check the number of fields and data type of each field it receives, if it expects to receive datums of a specified structure and data type.
- If feasible, the minnow should be coded in a general manner to work with different kinds of datums and data types, so that it can be used in a variety of PHISH input scripts

* Which port a datum arrived on is the only attribute of a received datum that a minnow can query (other than the format and content of the datum itself); see the phish.datum function. It cannot query which minnow sent it via what output port or which connection to the input port it arrived by. This is because these are really settings determined by the PHISH input script, and the minnow should not depend on them. If such info is really necessary for the minnow to know, then it can be encoded as a field in the datum itself, so the minnow can extract it.

Here are other flexible attributes of input and output ports to note, all enabled by the hook command in a PHISH input script:

* A single input port can receive datums from multiple other schools of minnows and multiple output ports.
* A single output port can send datums to multiple other schools of minnows and multiple input ports. This means an individual datum may be sent multiple times to different minnows.
* A minnow can send datums via its output port to its own input port.

All of these scenarios can be setup by appropriate use of hook commands in a PHISH input script.

An additional issue to consider is whether a communication channel can be saturated or drop datums. Imagine a PHISH net where one minnow sends datums at a high rate to a receiving minnow, which cannot process them as fast as they are sent. Over time, the receiving minnow is effectively a bottleneck in processing a stream of data. The PHISH library will not lose messages in this scenario, rather the overall processing pipeline naturally throttles itself to the rate of the bottlenecking minnow. This is handled by the underlying MPI or socket message passing protocols. ZMQ handles this naturally. In the case of MPI, the sending and receiving processes coordinate data exchanges. By default this is done via MPI_Send() and MPI_Recv() calls. If you get a run-time MPI error about dropping messages, then you should use occasionally use the "safe" mode of data exchange which can be enabled by the set safe command in a PHISH input script or "--set safe" command-line option. This will use MPI_Ssend() calls which enforce extra handshaking between the sending and receiving processes to avoid dropping messages.

3.4 Shutting down a minnow

PHISH minnows can be designed to process a finite or infinite stream of data. In the infininte case, the PHISH net of minnows is typically shut down by the user killing one or more of the processes. Note that the current ZMQ version of PHISH cannot guarantee that all processes will be shut-down cleanly. You may need to kill some of the procsses manually.

In the finite case, you typically want each minnow in the net to shut down cleanly.

The PHISH library sends special "done" messages when a minnow closes one of its output ports. This is triggered by a call to the phish_close function, which closes a single port, or the phish_exit function which closes all output ports. A "done message is sent to each receiving minnow of each input port connected to the corresponding output port. The receiving minnow counts these messages as they arrive. When it has received one "done" message from every minnow that connects to one of its input ports, it closes the input port and the library calls back to the minnow (if a callback function was defined by the phish_input function). When all its input ports have been closed it makes an additional callback to the minnow (if a callback function was defined by the phish_callback function).

This mechanism is often sufficient to trigger an orderly shutdown of an entire PHISH net by all its minnows, if the most upstream minnow initiates the process by closing its output ports via a call to phish_exit. Exceptions are when a school of minnows exchanges data in a "ring" style of commuication as setup by the hook ring command in a PHISH input script.

In case of the ring, if the first minnow in the ring invokes the phish_exit function, it will no longer be receiving datums when the last minnow in the ring attempts to send it a "done" message. In this case, the first minnow should instead invoke phish_close on the output port for the ring, then wait to receive its final "done" message before calling phish_exit.

Another exception is when minnows send datums to themselves in a looping fashion. In this case, you typically to include code in callback functions invoked when "done" messages are received to handle the shutdown logic. See the minnow/tri.py for an example of how this can be done.

3.5 Building minnows

Minnows are stand-alone programs which simply need to be linked with the PHISH library. New single-file minnows written in C or C++ can be added to the minnow directory of the PHISH distribution and built in the following manner; minnows written in Python do not need to be built.

The easiest way to build all of PHISH, including the PHISH minnows, is to use the cross-platform CMake build system. We recommend building PHISH with a separate build directory:

```
$ tar xzvf phish.tar.gz -C ~/src
$ mkdir ~/build/phish
$ cd ~/build/phish
$ ccmake ~/src/phish-14sep12
```

Then, in the CMake curses interface, configure the build, generate makefiles, and build phish:

```
$ make
```

Note that if you add a new minnow to the minnow directory, simply re-run ccmake regenerate makefiles, and build - your minnow will automatically be incorporated into the build.

Alternatively, typing the following from the minnow directory will build all C and C++ minnows:

```
make machine
```

where machine is the suffix of one of the provided Makefiles, e.g. linux.mpi or linux.zmq. Type "make" to see a list of the different files and what compiler and MPI options they support.

The ".mpi" or ".zmq" suffix of the make target and associated Makefile refer to which version of the PHISH library will be linked against, either the MPI or ZMQ version.

The make command also builds non-PHISH C or C++ programs which are intended to be wrapped with one of the "wrap" minnows discussed above so they can be used as a minnow. Examples are the echo and reverse programs in the minnow directory.

If none of the provided Makefiles are a match to your machine, you can use one of them as a template and create your own. Note that only the top section for compiler/linker settings need be edited. This is where you should specify your compiler and linker and any switches they use. For the LIB setting, be sure to use the appropriate version of the PHISH library you are linking to, i.e. libphish-mpi.so or libphish-zmq.so.

IMPORTANT NOTE: When adding a new minnow that is a single file to the minnow directory, you should insure the string "MINNOW" appears somewhere in the *.cpp or *.c file. This is how the top-level minnow/Makefile includes it in the build list. It will then be automatically built with the other minnows.

IMPORTANT NOTE: If you wish to switch the PHISH library used with your minnows (MPI vs ZMQ), you should type "make clean-all" and then re-compile and re-link all the minnows. You can also type "make clean" to simply delete all object files.

If your new minnow is complex enough to consist of multiple files, you can add a specific rule for how to build it to the Makefile.machine you use, e.g. that defines a new target with a list of OBJ files that it depends on. Or you can build it in a separate directory with your own custom Makefile, so long as you link to the PHISH library, similar to how the Makefiles in the minnow directory perform the final link step.

Your executable minnow files do not need to be added to the minnow directory. See the --path command-line switch for the bait.py tool for how to access minnows from other directories when running a PHISH net.

4. PHISH Library

This sections documents the API to the PHISH library that PHISH minnows call. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

The API for the MPI and ZMQ (socket) versions of the PHISH library are identical.

A general discussion of how and when minnows call PHISH library functions is given in the Minnows section of the manual.

The PHISH library has a C-style API, so it is easy to write minnows in any language, e.g. C, C++, Fortran, Python. A true C++-style API is also provided, which means a C++ program can use either the C or C++ API. A Python wrapper on the C-style API is also provided, which has a slightly different syntax for some functions. The doc pages for individual library functions document all 3 APIs. See the section below entitled C vs C++ vs Python interface for a quick overview.

PHISH minnows communicate with other minnows by sending and receiving datums. Before looking at individual library calls, it may be helpful to understand how data is stored internally in a datum by the PHISH library. This topic is discussed below, in the section entitled Format of a datum.

4.1 List of library functions

The PHISH library is not large; there are only a handful of calls, that can be grouped into the following categories. Follow the links to see a doc page for each library call.

1. Library calls for initialization
   phish_init()
   phish_input()
   phish_output()
   phish_callback()
   phish_check()
2. Library calls for shutdown
   phish_exit()
   phish_close()
3. Library calls for receiving datums
   phish_loop()
   phish_probe()
   phish_recv()
   phish_unpack()
   phish_datum()
4. Library calls for sending datums
   phish_send()
   phish_send_key()

4.2 Building the PHISH library

There are two different versions of the PHISH library that can be built. One that calls message-passing functions from the MPI library, and one that calls socket functions from the ZMQ library. In either case, the library should typically be built as a shared library so it can loaded at run-time by each minnow. This is required if the minnow is written in Python.

The easiest way to build all of PHISH, including the PHISH libraries, is to use the cross-platform CMake build system. We recommend building PHISH with a separate build directory:

```
$ tar xzvf phish.tar.gz -C ~/src
$ mkdir ~/build/phish
$ cd ~/build/phish
$ ccmake ~/src/phish-14sep12
```

Then, in the CMake curses interface, configure the build, generate makefiles, and build phish:

```
$ make
```

Alternatively, you can build either version from the src directory of the distribution by typing one of these lines:

```
make -f Makefile.machine mpi
make -f Makefile.machine zmq
```

where "machine" is the name of one of the Makefiles in the directory. These should produce the file libphish-mpi.so or libphish-zmq.so.

If none of the provided Makefiles are a match to your machine, then you can use of them as a template for creating your own, e.g. Makefile.foo. Note that only the top section for compiler/linker settings need be edited. This is where you should specify your compiler and any switches it uses. The MPI_INC setting is only needed if you are building the MPI version of the library, and the compiler needs to know where to find the mpi.h file. Likewise the ZMQ_INC setting is only needed if you are building the ZMQ version of the library, and the compiler needs to know where to find the zmq.h file. The MPI_LIB and ZMQ_LIB settings are for the MPI and ZMQ library themselves and any other auxiliary libraries they require.

If the build is successful, a libphish-mpi.a or libphish-zmq.a file is produced.

You can also type

```
make -f Makefile.machine clean
```

to remove *.o and lib*.so files from the directory.

4.3 C vs C++ vs Python interface

As noted above, the APIs to the PHISH library for C versus C++ versus Python are very simliar. A C++ program can use either the C or C++ API.

To use the C interface, a C or C++ program includes the file src/phish.h and makes calls to functions as follows:

```
#include "phish.h"
phish_error("My error");
```

The C++ interface in src/phish.hpp encloses the PHISH API in the namespace "phish", so functions can be invoked as

```
#include "phish.hpp"
phish::error("My error");
```

or as

```
#include "phish.hpp"
using namespace phish
error("My error");
```

To use the Python interface, see this section of the manual for details. A Python program can invoke a library function as

```
import phish
phish.error("My error")
```

or

```
from phish import *
error("My error")
```

4.4 Format of a datum

The chief function of the PHISH library is to facilitate the exchange of data between minnows. This is done through datums, which contain one or more fields. Each field is a fundamental data type such as a "32-bit integer" or a "vector of doubles" or a NULL-terminated character string.

The PHISH library defines a specific explicit type for each fundamental data type it recognizes, such as "int32" for 32-bit signed integers, or "uint64" for 64-bit unsigned integers, or "double" for a double-precision value. This is so that the format of the datum, at the byte level, is identical on different machines, and datums can thus be exchanged between minnows running on machines with different word lengths or between minnows written in different languages (e.g. C vs Fortran vs Python).

IMPORTANT NOTE: Different endian ordering of fundamental numeric data types on different machines breaks this model. We may address this at some future point within the PHISH library.

This is the byte-level format of datums that are sent and received by minnows via the PHISH library:

- # of fields in datum (int32_t)
- type of 1st field (int32_t)
- size of 1st field (optional int32_t)
- data for 1st field (bytes)
- type of 2nd field (int32_t)
- size of 2nd field (optional int32_t)
- data for 2nd field (bytes)
- ...
- type of Nth field (int32_t)
- size of Nth field (optional int32_t)
- data for Nth field (bytes)

Integer flags are interleaved with the fundamental data types and the flags themselves are all 32-bit signed integers. This allows minnows that call the phish_pack and phish_unpack functions to use the usual C "int" data type as function arguments, instead of the int32_t types defined in the function prototypes. The compiler will only give an error if the native "int" on a machine is not a 32-bit integer. See the doc pages for phish_pack and phish_unpack for details.

The "type" values are one of these settings, as defined in src/phish.h:

- PHISH_CHAR = 0
- PHISH_INT8 = 1
- PHISH_INT16 = 2
- PHISH_INT32 = 3
- PHISH_INT64 = 4
- PHISH_UINT8 = 5
- PHISH_UINT16 = 6

- PHISH_UINT32 = 7
- PHISH_UINT64 = 8
- PHISH_FLOAT = 9
- PHISH_DOUBLE = 10
- PHISH_RAW = 11
- PHISH_STRING = 12
- PHISH_INT8_ARRAY = 13
- PHISH_INT16_ARRAY = 14
- PHISH_INT32_ARRAY = 15
- PHISH_INT64_ARRAY = 16
- PHISH_UINT8_ARRAY = 17
- PHISH_UINT16_ARRAY = 18
- PHISH_UINT32_ARRAY = 19
- PHISH_UINT64_ARRAY = 20
- PHISH_FLOAT_ARRAY = 21
- PHISH_DOUBLE_ARRAY = 22
- PHISH_PICKLE = 23

PHISH_CHAR, PHISH_INT*, PHISH_UINT*, PHISH_FLOAT, and PHISH_DOUBLE are a single character, a signed integer (of length 8,16,32,64 bits), an unsigned integer (of length 8,16,32,64 bits), a float (typically 4 bytes), and a double (typically 8 bytes).

PHISH_RAW is a string of raw bytes which minnows can format in any manner, e.g. a C data structure containing a collection of various C primitive data types. PHISH_STRING is a standard C-style NULL-terminated C-string. The NULL is included in the field.

The ARRAY types are contiguous sequences of int*, uint*, float, or double values, packed one after the other.

PHISH_PICKLE is an option available when using the Python wrapper on the PHISH library to encode arbitrary Python objects in pickled form as a string of bytes.

The "size" values are only included for PHISH_RAW (# of bytes), PHISH_STRING (# of bytes including NULL), the ARRAY types (# of values), and PHISH_PICKLE (# of bytes).

The field data is packed into the datum in a contiguous manner. This means that no attention is paid to alignment of integer or floating point values.

The maximum allowed size of an entire datum (in bytes) is set to a default value of 1024 bytes or 1 Kbyte. This can be overridden via the set memory command in a PHISH input script or "--set memory" command-line option.

When a datum is sent to another minnow via the MPI version of the PHISH library, MPI flags the message with an MPI "tag". This tag encodes the receiving minnow's input port and also a "done" flag. Specifically, if the datum is not a done message, the tag is the receiver's input port (0 to Nport-1). For a done message a value of MAXPORT is added to the tag. See the discussion of MAXPORT in this section of the manual.

Similarly, the ZMQ version of the PHISH library prepends a "done" flag and port number to each datum.

See the phish_input doc page for a discussion of ports. See the shutdown section of the Minnows doc page for a discussion of "done" messages.

5. Examples

This is the list of PHISH input scripts provided in the example directory of the distribution. Many of them use minnows which come in 2 flavors in the minnow directory, written in C++ or Python. The scripts can thus be edited to use all C++ or all Python minnows or a mixture of the two in the same PHISH net. See the comment lines at the top of each script for instructions on what variables can be defined via command-line arguements to the bait.py tool.

- in.filelist
- in.hook
- in.slow
- in.wordcount
- in.wrapsink
- in.wrapsource
- in.wrapsourcefile
- in.wrapss

---

in.filelist

This script simply prints a list of files. It uses the filegen and print minnows.

It can process a large corpus of files by specifying one or more directory names as arguments to the filegen minnow via the variable "files".

Here is an example of how to run the script with bait.py:

```
python ../bait/bait.py -v files in.* -p ../minnow <in.filelist
```

in.hook

This script generates a diagram of the various styles of hook connections that can be made between schools of minnows via the bait.py hook command. It is not meant to be run as a parallel program, only processed by the bait.py tool.

Here is an example of how to run the script with bait.py:

```
python ../bait/bait.py -b graphviz <in.hook
```

The resulting dot file can be visualized via the GraphViz tools.

in.slow

This script prints words from files while using a slowdown minnow to make it operate more slowly. It uses the filegen, file2words, slowdown, and print minnows.

See the input script for variables that can be set, including the delay invoked by the slowdown minnow via the variable "delta".

Here is an example of how to run the script with bait.py:

```
python ../bait/bait.py -v files in.* -v delta 1.0 -p ../minnow <in.slow
```

---

in.wordcount

This script performs a word frequency count across a set of input text files, similar to a MapReduce operation. It uses the filegen, file2words, count, sort, and print minnows, connected as diagramed in this section of the Introduction.

See the input script for variables that can be set, including the number of file2words and count minnows for parallelism. The script can process a large corpus of files by specifying one or more directory names as arguments to the filegen minnow via the variable "files".

Here is an example of how to run the script with bait.py:

```
python ../bait/bait.py -v files in.* -p ../minnow <in.wordcount
```

---

in.wrapsink

This script demonstrates the use of the wrapsink minnow which is used to wrap a non-PHISH application. Wrapsink passes string datums it receives to any program thru stdin. The script uses the reverse program in the minnow dir to reverse the name of each file generated by the filegen minnow.

Here is an example of how to run the script with bait.py:

```
python ../bait/bait.py -v files in.* -p ../minnow <in.wrapsink
```

---

in.wrapsource

This script demonstrates the use of the wrapsource minnow which is used to wrap a non-PHISH application. Wrapsource invokes the application and reads its output from stdout, a line at a time. The line is converted to a string datum which is sent downstream to other minnows. The script uses the shell command "ls *" as its program and sends its output to the print minnow. Thus a list of filenames in the directory is generated.

Here is an example of how to run the script with bait.py:

```
python ../bait/bait.py -p ../minnow <in.wrapsource
```

---

in.wrapsourcefile

This script demonstrates the use of the wrapsource minnow with its optional file input, which is used to wrap a non-PHISH application. Wrapsource takes filenames as received datums, and invokes the application for each one, then reads its output from stdout, a line at a time. The line is converted to a string datum which is sent downstream to other minnows. The script uses the shell command "wc %s" as its program, where the "%s" represents the filename that is substituted for each time the program is invoked. It sends its output to the print minnow. Thus a list of filename sizes is generated.

See the input script for variables that can be set, including the number of wrapsource minnows for parallelism. The script can process a large corpus of files by specifying one or more directory names as arguments to the filegen minnow via the variable "files".

Here is an example of how to run the script with bait.py:

```
python ../bait/bait.py -p ../minnow -v files in.* <in.wrapsourcefile
```

in.wrapss

This script demonstrates the use of the wrapss minnow which is used to wrap a non-PHISH application. Wrapss takes string datums and passes them to the stdin as lines of input to a running application. It also polls the application for output written to stdout, which it reads a line at a time. The line is converted to a string datum which is sent downstream to other minnows. The script uses the echo or reverse programs in the minnow dir as the application. Thus a list of filenames is printed, in reverse order if desired.

The script can process a large corpus of files by specifying one or more directory names as arguments to the filegen minnow via the variable "files".

Here is an example of how to run the script with bait.py:

```
python ../bait/bait.py -v files in.* -p ../minnow <in.wrapss
```

6. Python Interface to PHISH

A Python wrapper for the PHISH library is included in the distribution. This allows a minnow written in Python to call the PHISH library. The advantage of using Python is how concise the language is, enabling rapid development and debugging of PHISH minnows and nets. The disadvantage is speed, since Python is slower than a compiled language and there is an extra layer of callback functions between C++ and Python, when receipt of a datum makes a callback to a minnow written in Python.

Before using the PHISH library in a Python script, the Python on your machine must be able to find the PHISH library and wrapper. This is discussed below.

The Python interface to the PHISH library is very similar to the C interface. See this section of the doc pages for a brief overview. Individual library function doc pages give examples of how to use the Python interface.

Extending Python with the PHISH library

For a Python minnow to use the PHISH library, it must find two files at run-time that are part of PHISH. The first is the PHISH wrapper. The second is the PHISH library. It must also be able to find other shared libraries that the PHISH library depends on, e.g. MPI or ZMQ libraries, which is discussed in the next section.

There are two different ways to enable Python to find the two PHISH files.

(1) Add two lines to your shell start-up script.

For csh or tcsh, add lines like these to your .cshrc file:

```
setenv PYTHONPATH $PYTHONPATH:/home/sjplimp/phish/python
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/sjplimp/phish/src
```

For bash, add lines like these to your .bashrc file:

```
export PYTHONPATH $PYTHONPATH:/home/tshead/src/phish/python
export LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/tshead/build/phish/src
```

After editing your shell start-up script, be sure to invoke it, e.g. source .cshrc.

Note: On OSX systems, use DYLD_LIBRARY_PATH instead of LD_LIBRARY_PATH.

(2) Add the Python wrapper to the site-packages directory of your installed Python and the PHISH library to a directory the system looks in to load shared libraries.

The site-packages dir is typically something like /usr/lib/python/site-packages if you are using the system Python, or /usr/local/lib/python/site-packages if you installed Python yourself.

Lines like these will copy the needed files:

```
% cp -r /home/sjplimp/phish/python/phish /usr/local/lib/python/site-packages
% cp /home/sjplimp/phish/src/*.so /usr/local/lib
```

The latter command will copy all the PHISH shared libraries you have built, including the bait.py tool backends. The latter is necessary for using bait.py.

You will need to prefix the lines with "sudo" if you need permission to copy into directories owned by root.

Creating a shared MPI or ZMQ library

To use the MPI or ZMQ version of the PHISH library from a Python minnow, a shared-library version of MPI or ZMQ must also exist, in a place the system can find it. On Linux this is a library file that ends in ".so", not ".a". Such a shared library may not normally not built if you installed MPI or ZMQ yourself, but it is easy to do. Here is how to do it for MPICH, a popular open-source version of MPI, distributed by Argonne National Labs. From within the mpich directory, type

```
./configure --enable-sharedlib=gcc
make
make install
```

You may need to prepend "sudo" to the last line. The end result should be the file libmpich.so put into /usr/local/lib. As an alternative to the final make install, you can add the directory the libmpich.so file is in to your LD_LIBRARY_PATH environment variable, as illustrated above.

To build ZMQ as a shared library, you may need to specify --enable-shared during the configuration process, which is the default.

Testing the PHISH library from Python

To test if your Python can find all the files it needs to use the PHISH library, launch python and type:

>>> import phish

If you don't get an error, you're good to go.

7. Errors

This section discusses error and warning messages generated by the PHISH library and bait.py and bait.py tools. It also gives tips on debugging the operation of PHISH nets.

Debugging PHISH nets

A PHISH net can be difficult to debug because it may involve many independent running processes exchanging datums in a complex pattern, and rapid processing of large volumes of data. Here are some ideas that may be helpful in finding a bug if something goes wrong:

- As with any parallel program, running on as few processors as possible (that still exhibit the bug) simplifies debugging. For a PHISH net, each minnow is a process, but you may be able to reduce the minnow count via the school command in the PHISH input script.
- In principle, a PHISH net should behave similarly whether it is run entirely on a desktop machine, on a large parallel machine, or on a distributed network. Running all minnows on a single desktop machine is almost always the easiest mode to debug in.
- One or more slowdown minnows can be inserted in a PHISH net to slow down the rate at which datums are processed.
- The print minnow can receive the output of any minnow and print it to the screen. This is done by using the hook command in a PHISH input script. The output of a minnow can be connected to a print minnow in addition to other minnows. Also note that hooking a minnow's output port to another input port is optional, as discussed in this section. This means that you can often debug a PHISH net in stages by commenting out downstream minnows and then turning them on one at a time.
- Printf() or fprintf() statements can be added to a minnow's source code to print messages to the screen or a file when datums are received, processed, or sent. Doing this stage by stage, beginning with the first datums read or generated by the PHISH input script, is an effective way to verify that datums are formatted correctly, and are being sent and received as expected.

Error and warning messages from the PHISH library

When a minnow makes a call to the PHISH library, various error and warning conditions are checked for. If an error is encountered, a message in the following format is printed to stderr:

```
PHISH ERROR: Minnow executable ID school-id # global-id: message
```

where "executable" is the name of executable minnow file (not the full path, just the filename), "school-id" is the ID of the minnow as specified in the PHISH input script, "global-id" is the global-ID of the minnow, and message is the error message. Each minnow has a global ID from 0 to Nglobal-1, where Nglobal is the total number of minnows in the net specified by the PHISH input script. This supplementary information is helpful in debugging which minnow generated the error message.

If a warning condition is detected, a similarly-formatted message is printed, with ERROR replaced by WARNING.

Note that by default, an error condition will cause an abort, shutting-down the entire PHISH net. However, it is possible for a minnow to explicitly cancel the abort. In this case, the PHISH library call that triggered the error will return a non-zero integer error code to indicate that an error occurred. See the documentation for phish_abort() for more information on how to cancel an abort.

The messages should be self-explanatory. See the doc page for the individual">>PHISH library, or the doc pages for PHISH library functions for relevant details. If necessary, the library source code in src/phish-common.cpp or src/phish-mpi.cpp or src/phish-zmq.cpp can be searched for the message text.

Error messages from bait.py

The bait.py tool is used to process PHISH input scripts and optionally run them. Any errors that bait.py encounters in the input script generate error messages that should be self-explanatory.

See the doc page for the individual">>bait.py tool, or the doc pages for PHISH input script commands for relevant details. If necessary, the Python source code in bait/bait.py can be searched for the message text.

hook command

Syntax:

hook sendID:outport style recvID:inport

- sendID = ID of minnows which will send datums
- outport = output port datums are written to by sending minnows (default = 0)
- style = communication pattern between sending and receiving minnows
- minnows = *single* or *paired* or *hashed* or *roundrobin* or *direct* or *bcast* or *chain* or *ring* or *publish* or *subscribe*
- recvID = ID of minnows which will receive datums
- inport = input port datums are read from by receiving minnows (default = 0)

Examples:

```
hook 1 single 2
hook 2:1 hashed 2:1
```

Description:

Hook is a command that can be used in a PHISH input script which is recognized by the bait.py setup program. It determines how the output from a minnow in one school is routed to the input of a minnow in another school when the PHISH program is run. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The topology of communication patterns between minnows defined by a series of hook commands defines how multiple schools of minnows are harnessed together to perform a desired computational task. It also defins how parallelism is exploited by the schools of minnows.

A hook is made between two schools of minnows, one school sends datums, the other set receives them. Each school may contain one or more minnows, as defined by the school command. Since a datum is typically sent from a single minnow to a single receiving minnow, the style of the hook determines which minnow in the sending schoold communicates with which minnow in the receiving school.

Each minnow can send datums through specific output ports. If a minnow defines N output ports, then they are numbered 0 to N-1. Likewise a minnow can receive data through specific input ports. If a minnow defines M input ports, then they are numbered 0 to M-1. Ports enable a minnow to have multiple input and output hooks, and for a PHISH input script to hook a single set of minnows to multiple other sets of minnows with different communication patterns. For example, a stream of data might be processed by a minnow, reading from its input port 0, and writing to its output port 0. But the minnow might also look for incoming datums on its input port 1, that signify some kind of external message from a "control" minnow triggered by the user, e.g. asking the minnow to print out its current statistics. See the Minnows doc page for more information about how minnows can define and use ports.

The specified *sendID* and *outport* are the minnows which will send datums through their output port *outport*. If *outport* is not specified with a colon following the *sendID*, then a default output port of 0 is assumed.

The specified *recvID* and *inport* are the minnows which will receive the sent datums through their input port *inport*. If *inport* is not specified with a colon following the *recvID*, then a default input port of 0 is assumed.

Both *sendID* and *recvID* must be the IDs of minnows previously defined by a minnow command.

Note that there can be multiple hook commands which hook the same *sendID* and same (or different) *outport* to different *recvID:inport* minnows. Likewise, there can be multiple hook commands which hook the same *recvID* and same (or different) *inport* to different *sendID:outport* minnows. There can even be multiple hook commands which hook the same *sendID* and same (or different) *outport* to the same *recvID:inport* minnows.

Also note that for all of the styles (except as noted below), the *sendID* and *recvID* can be the same, meaning a set of minnows will send datums to themselves.

These are the different hook styles supported by the hook command.

The *single* style hooks *N* sending minnows to one receiving minnow. *N* = 1 is allowed. All the sending minnows send their datums to a single receiving minnow.

The *paired* style hooks *N* sending minnows to *N* receiving minnows. *N* = 1 is allowed. Each of the *N* sending minnows sends it datums to a specific partner receiving minnow.

The *hashed* style hooks *N* sending minnows to *M* receiving minnows. *N* does not have to equal *M*, and either or both of *N*, *M* = 1 is allowed. When any of the *N* minnows sends a datum, it must also define a value for the PHISH library to hash on, which will determine which of the *M* receiving minnows it is sent to. See the doc page for the phish_send_hashed() library function for more explanation of how this is done.

The *roundrobin* style hooks *N* sending minnows to *M* receiving minnows. *N* does not have to equal *M*, and either or both of *N*, *M* = 1 is allowed. Each of the *N* senders cycles through the list of *M* receivers each time it sends a datum, in a roundrobin fashion. a different. If the receivers are numbered 0 to M-1, a sender will send its first datum to 0, its 2nd to 1, its Mth to M-1, its M+1 datum to 0, etc.

The *direct* style hooks *N* sending minnows to *M* receiving minnows. *N* does not have to equal *M*, and either or both of *N*, *M* = 1 is allowed. When any of the *N* minnows sends a datum, it must also choosed a specific one of the *M* receiving minnows to send to. See the doc page for the phish_send_direct() library function for more explanation of how this is done.

The *bcast* style hooks *N* sending minnows to *M* receiving minnows. *N* does not have to equal *M*, and either or both of *N*, *M* = 1 is allowed. When any of the *N* minnows sends a datum, it sends a copy of it once to each of the *M* receiving minnows.

The *chain* style configures *N* minnows as a 1-dimensional chain so that each minnow sends datums to the next minnow in the chain, and likewise each minnow receives datums from the previous minnow in the chain. The first minnow in the chain cannot receive, and the last minnow in the chain cannot send. *N* > 1 is required. The *sendID* must also be the same as the *recvID*, since the same set of minnows is sending and receiving.

The *ring* style is the same as the *chain* style, except that the *N* minnows are configured as a 1-dimensional loop. Each minnow sends datums to the next minnow in the loop, and likewise each minnow receives datums from the previous minnow in the loop. This includes the first and last minnows. *N* > 1 is required. The *sendID* must also be the same as the *recvID*, since the same set of minnows is sending and receiving.

The following hook styles will be supported in future versions of PHISH:

The *publish* and *subscribe* styles are different in that they do not hook two sets of minnows to each other. Instead they hook one set of minnows to an external socket, either for writing or reading datums. The external socket will typically be driven by some external program which is either reading from the socket or writing to it, but the running PHISH program requires no knowledge of that program. It could be another PHISH program or some completely different program.

The *publish* style hooks *N* sending minnows to a socket. *N* = 1 is allowed. The *recvID:inport* argument is replaced with a TCP port #, which is an integer, e.g. 25. When each minnow sends a datum it will "publish" the bytes of the datum to that TCP port, on the machine the minnow is running on. In socket lingo, "publishing" means that the sender has no communication with any processes which may be reading from the socket. The sender simply writes the bytes and continues without blocking. If no process is reading from the socket, the datum is lost.

The *subscribe* style hooks *M* receiving minnows to a socket. *M* = 1 is allowed. The *sendID:outport* argument is replaced with a hostname and TCP port #, separated by a colon, e.g. www.foo.com:25. Each minnow receives datums by "subscribing" to the TCP port on the specified host. In socket lingo, "subscribing" means that the receiver has no communication with any process which is writing to the socket. The receiver simply checks if a datum is available and reads it. If a new datum arrives before the receiver is ready to read it, the datum is lost.

Note that multiple processes can publish to the same physical socket, and likewise multiple processes can subscribe to the same physical socket. In the latter case, each receiving process reads the same published datum.

Restrictions:

The *publish* and *subscribe* styles have not been implemented yet by any of the PHISH library versions.

Related commands:

minnow, school

Default: none

minnow command

Syntax:

```
minnow ID exefile arg1 arg2 ...
```

- ID = ID of minnow
- exefile = executable file name
- arg1,arg2 ... = arguments to pass to executable

Examples:

```
minnow 1 count
minnow 5 filegen ${files}
minnow myapp app 3 f1.txt 4.0
```

Description:

Minnow is a command that can be used in a PHISH input script which is recognized by the bait.py setup program. It defines a minnow application and assigns it an ID which can be used elsewhere in the input script. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The *ID* of the minnow can only contain alphanumeric characters and underscores.

The *exefile* is the name of the executable which will be launched when the PHISH program is run. It should reside in one of the directories specified by the *-path* command-line argument for bait.py.

The *arg1*, *arg2*, etc keywords are arguments that will be passed to the *exefile* program when it is launched.

Restrictions: none

Related commands:

school

school command

Syntax:

```
school minnow-ID Np keyword value ...
```

- minnow-ID = ID of minnow
- Np = # of instances of this minnow to launch
- zero or more keyword/value pairs can be appended

```
possible keywords = bind
  bind values = N1,C1 N2,C2 ...
    N1,C1 = node ID, core ID for first minnow
    N2,C2 = node ID, core ID for second minnow, etc
    see discussion below for wildcard format
```

Examples:

```
school 3 10
school countapp 1
school countapp 1 host foo.locallan.gov
school myApp 5 bind *,0
school myApp 2 bind 0,0 3,2
school myApp 5 bind 0*1,* 2,0
```

Description:

School is a command that can be used in a PHISH input script which is recognized by the bait.py setup program. It determines how a minnow application will be launched when the PHISH program is run. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The *minnow-ID* is the ID of the minnow, as previously defined by a minnow command.

Np is the number of instances of this minnow that will be launched when the PHISH program is run.

The *bind* keyword allows you to control what machine or what nodes and cores of a multi-core machine that each instance of a minnow will run on.

There are 3 ways to do this assignment in PHISH; each is discussed below.

- via the mpirun command (only possible for the MPI backends of bait.py)
- via the *bindorder*, *pernode*, and *numnode* settings of the bait.py set command or bait.py *--set* or *-s* command-line switches
- via the *bind* options of the bait.py school command, as illustrated above

The examples below use a PHISH input script with 2 schools of minnows, the first with a school *Np* setting of 12, the second with 8, for a total of 20 minnows or processes.

Here is how binding works for the MPI backends to the bait.py command.

If *bindorder* is unset (or set to 0) and no *bind* keywords are used with the school command, then minnows will be assigned to the nodes by the mpirun command. By default this is typically first by core, then by node. E.g. on a machine with quad-core nodes, the 12 instances of the first minnow would run on the 12 cores of the first 3 nodes, and the 8 instances of the second minnow on the 8 cores of the last 2 nodes, since your job will be allocated 5 nodes when mpirun requests 20 processes for the entire PHISH program. The mpirun commands for different versions of MPI have options that can be used to control the assignment, e.g. to assign first by node, then by core. See the man pages for mpirun for details.

The other 2 methods of assignment will override any options to the mpirun command.

If *bindorder* is set to 1 or 2, and no *bind* keywords are used with the school command, then minnows are assigned to cores in the following manner, using the *pernode* and *numnode* settings of the bait.py set command.

If *bindorder* is set to 1, then minnows are assigned in a double loop, with the inner loop over cores from 0 to pernode-1 and the outer loop over nodes from 0 to numnode-1. E.g. on a machine with quad-core nodes, the 12 instances of the first minnow would run on the 12 cores of the first 3 nodes, and the 8 instances of the second minnow on the 8 cores of the last 2 nodes, since your job will be allocated 5 nodes when mpirun requests 20 processes for the entire PHISH program. This assumes that you have set *pernode* to 4 and *numnode* to 5; the latter is the default.

If *bindorder* is set to 2, then minnows are assigned in a double loop, with the inner loop over nodes from 0 to numnode-1 and the outer loop over cores from 0 to pernode-1. E.g. on a machine with quad-core nodes, the 12 instances of the first minnow would be spread across all 5 nodes (3 on the first 2, 2 on the last 3), as would the 8 instances of the second minnow (1 on the last 2, 2 on the last 3), since your job will be allocated 5 nodes when mpirun requests 20 processes for the entire PHISH program. This again assumes that you have set *pernode* to 4 and *numnode* to 5.

But if you were allocated 12 nodes and are only running 20 minnows, you could set *numnode* to 12 and *bindorder* to 2. The 12 instances of the first minnow would be spread across all 12 nodes (1 each), and the 8 instances of the second minnow would be spread across the first 8 nodes (1 each).

If the *bind* keyword is used with any school command, it must be used with all of them. If it is used, then each minnow is assigned explicitly to a specific node and core, so that the methods of assignment just described are overridden. However if wildcards are used in the explicit assignments, then the *bindorder*, *pernode*, and *numnode* settings are used, as explained below.

The *bind* keyword takes one or more node/core ID pairs as values. Node IDs must be from 0 to *numnode*-1 inclusive. Core IDs must be from 0 to *pernode*-1 inclusive. Each node and core ID can represent a range of consecutive node and core IDs if it is specified using a wildcard. This takes the form "*" or "*n" or "n*" or "m*n". If N = *numnode* or *pernode* for node or core count, then an asterisk with no numeric values means all IDs from 0 to N-1 (inclusive). A leading asterisk means all IDs from 0 to n (inclusive). A trailing asterisk means all IDs from n to N-1 (inclusive). A middle asterisk means all IDs from m to n (inclusive). Specifying an ID that is < 0 or >= N is an error.

For each *bind* value an ordered list of explicit node/core IDs is generated, expanding each value with wildcards as needed. If both the node and core ID have a wildcard then the value is expanded in a double loop. The ordering of the double loop is controlled by the *bindorder* setting as explained above: inner/outer = core/node for bindorder 1, inner/outer = node/core for bindorder 2.

For example, on a machine with 4 cores per node (*pernode* = 4), and 3 nodes allocated for your PHISH run (*numnode* = 3), this command

```
school myApp 5 bind 1*,* 0,2*
```

would generate the following list of 10 node/core ID pairs:

```
(1,0), (1,1) (1,2) (1,3) (2,0) (2,1) (2,2) (2,3) (0,2) (0,3)
```

The minnow instances are assigned to this list in order. I.e. the first minnow instance will run on the 1st node/core ID, the 2nd instance of the minnow on the 2nd node/core ID, etc.

If the number of instances Ni < the length of the list, then only the first Ni node/core ID pairs are used. If Ni > length of the list, then the list is looped over until all minnow instances are assigned. Note that this can result in multiple minnows being assigned to the same core.

Binding for the ZMQ backends to the bait.py command works the same way as for the MPI backends, with 3 differences.

(1) The first method described above, i.e. letting the mpirun command assign minnows to physical processors, is not an option. One of the other 2 methods must be used.

(2) Once a minnow has been assigned to a node ID and core ID, the core ID is ignored.

(3) The node ID is converted to a machine hostname. The set of possible hostnames is determined by the variable hostnames command which must be specified either in the input script or as a command-line option to the bait.py tool. As described on the variable doc page, a hostname can be a machine name (foo.localnet.gov) or a node name on a parallel machine (rs2001).

The ZMQ backend launches each minnow on a specific hostname. If the host is a multi-core node, then it may launch multiple minnows on the node, relying on the node operating system to distribute the minnow processes efficiently across cores.

As described above each node ID is a value N from 0 to *numnode*-1 inclusive. This value is used to index into the list of hostnames. If the list length L is smaller than N, then the index = N mod L. E.g. if the node ID N is 10 and the hostname list is of length 4, then the node maps to the 3rd hostname in the list (index = 2).

Restrictions: none

Related commands:

minnow, set

Default:

If a school command is not specified for a particular minnow, then Np is assumed to be 1, so that one instance of the minnow is launched when the PHISH program is run.

set command

Syntax:

set keyword value

- keyword = *memory* or *self* or *queue* or *safe* or *port* or *pernode* or *numnode* or *bindorder*

```
memory value = N
    N = max size of datum in Kbytes
  self value = S
    S = allow queueing of this many messages to self
  queue value = Q
    Q = allow queueing of this many previously received messages
  safe value = M
    M = send a datum "safely" every this many sends
  pernode value = Nc
    Nc = # of cores per node to use when binding minnows to processors
  numnode value = Nn
    Nn = # of nodes to use when binding minnows to processors
  bindorder value = B
    B = 0 = no bind order is imposed
    B = 1 = perform binding by looping over cores, then over nodes
    B = 2 = perform binding by looping over nodes, then over cores
```

high-water-mark value = N N = maximum number of outgoing messages to queue to a recipient

Examples:

```
set memory 1024
set self 20
set safe 10
set bindorder 2
```

Description:

Set is a command that can be used in a PHISH input script which is recognized by the bait.py setup program. It resets default values that are used by the bait.py program as it reads and processes commands from the PHISH input script.

The *memory* keyword sets the maximum length of datums that are exchanged by minnows when a PHISH program runs. Send and receive buffers for datums are allocated by the PHISH library. The *N* setting is in Kbytes, so that *N* = 1024 is 1 Mbyte, and *N* = 1048576 is 1 Gbyte. The default is *N* = 1, since typical PHISH minnows send and receive small datums.

The *self* keyword sets the size of a queue used to buffer datums sent by a minnow to itself. In most PHISH input scripts sending to self this never happens, but it is allowed. Since messages to self are received with the highest priority by PHISH library functions such as phish_loop() and phish_probe() that receive messages, it is also typical that this queue often needs to be no larger than S = 1 message, since $S refers to the maximum number of outstanding (sent, but not yet received) messages. But it can be set larger if a minnow may perform several sends (to itself) before receiving a new message.

The *queue* keyword sets the size of a queue used to buffer previously received datums when a minnow uses the phish_queue() and phish_dequeue() library calls, so that it can process a received datum later, after receiving other datums. The Q setting determines the maximum number of datums that can be queued at one time in this manner.

These 3 settings determine how much memory PHISH allocates. There are always 2 buffers of size *memory* = N, one for sending, one for receiving. There are at most *self* = S buffers, each of size N, allocated for self messages, and likewise at most *queue* = Q buffers, each of size N, allocated for queued messages. These buffers for self and queued messages are only allocated if they are needed as the PHISH script runs and messages are processed. Thus the overall memory footprint of the PHISH library is typically tiny, unless you set these parameters to large values.

The *safe* keyword is only relevant for the MPI version of the PHISH library. It causes the library to use a "safe" MPI_Ssend() call every Mth time it sends a datum, rather than the normal MPI_Send() function. If M = 0, "safe" sends are never performed.

Safe in this context refers to messages being dropped if the receiving process is backed up. This can happen if a minnow in a PHISH school of minnows is significantly slower to process datums than all the others, and a large number of datums are being continually sent to it. When a safe send is done, an extra handshake is performed between the sender and receiver to insure the receiver is ready for the datum. Doing this often enough should effectively throttle the incoming messages to a slow minnow, so an overflow does not occur. Note that the extra handshaking between the MPI processes slows down the rate at which small messages are exchanged, so you should not normally use this setting unless MPI errors arise, e.g. due to "too many unexpected messages". A good setting for M depends on how many minnows are sending to the overflowing minnow. Typical MPI implementations allow tens of thousands of small incoming messages to be queued, so M = 1000 or 10000 is fine if a single minnow is sending to the minnow. If 10 minnows are sending to the overflowing minnow, then M should be reduced by a factor of 10.

The *self* keyword is only relevant for the MPI version of the PHISH. A PHISH net described by an input script can include hook commands that route datums from a school back to itself, e.g. in some kind of looping fashion. This means that a procesor can send messages to itself. When this occurs the PHISH library uses a buffered MPI_Bsend() call. This avoids a possible hang due to a blocking send, which can happen in some MPI implementations. The value of *S* determines how big a buffer is reserved for the buffered send operations. It effectively means how many messsages (of maximum size, as determined by the memory keyword above), can be sent to self and buffered before they are received. In most cases the small default setting is sufficient, but if a minnow can send many messages to itself before reading and processing them, you may need to boost this setting to avoid an MPI error or hang.

The *pernode*, (numnode, and *bindorder* keywords are used to tailor the assignment of minnows to physical processors when the PHISH program is run. See the discussion of the *bind* option for the bait.py school command for more details on this topic.

Note that the default setting of *numnode* is equal to the total number of minnow instances *Np* in a PHISH input script, divided by the *pernode* setting. If *Np* is not evenly divisible by *pernode*, then *numnode* is incremented by one so that *numnode* * *pernode* >= *Np*.

The *high-water-mark* keyword only applies to the ZMQ version of PHISH. It sets the maximum number of outgoing messages that can be queued to a given recipient. Messages are queued when a recipient is receiving messages more slowly than the sender is sending them. Once the number of queued messages reaches the high water mark, the sender will block until the queue begins to empty. Note that messages are queued on a per-recipient rather than a per-port basis, so the memory used for queuing can vary dramatically depending on the size of the downstream school.

Restrictions: none

Related commands:

Any setting can be made either in a PHISH input script or via the *--set* command-line argument of the bait.py tool.

Default:

The default settings are memory = 1 (1 Kbyte), self = 8, queue = 8, safe = 0, pernode = 1, numnode = # of minnows / pernode, and bindorder = 0.

variable command

Syntax:

```
variable ID str1 str2 ...
```

Examples:

```
variable files f1.txt f2.txt f3.txt
variable N 100
```

Description:

Variable is a command that can be used in a PHISH input script which is recognized by the bait.py setup program. It creates a variable with name *ID* which contains a list of one or more strings. The variable can be used elsewhere in the input script. The substitution rules for variables are described by the bait.py doc page.

The *ID* of the variable can only contain alphanumeric characters and underscores. The strings can contain any printable character.

A variable can only be defined once, i.e. IDs must be unique. The one exception is for variables defined as command-line arguments to the bait.py command. In this case, a variable command in the input script is ignored if the variable was already set in the command line. This allows an input script to set default values for variables that can be overridden by command-line settings.

There is one variable that must be set when using the ZMQ backend to bait.py. Its ID is "hostnames" and it should be set to the list of hostnames on which to launch the PHISH job. If you are running on a large parallel machine, it is typically the names of the nodes assigned to your job by the batch allocator. If you are running on your desktop, it can be just the hostname of your machine. E.g.

```
variable hostnames node3 node4 node10
variable hostnames mybox
variable hostnames mybox otherbox.localnet.gov
```

This variable is used by the ZMQ backend to determine what host to launch each minnow on. See the bind option of the bait.py school command for further discussion on how to assign specific minnows to specific hosts.

Restrictions: none

Related commands:

Any variable can be set either in a PHISH input script or via the *--variable* or *-v* command-line arguments of the bait.py tool.

Default: none

count minnow

Syntax:

```
count
```

- this minnow takes no arguments

Examples:

```
count
```

Description:

Count is a PHISH minnow that can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The count minnow counts occurrences of strings it receives. When it shuts down it sends unique words and their associated counts.

Ports:

The count minnow uses one input port 0 to receive datums and one output port 0 to send datums.

Operation:

When it starts, the count minnow calls the phish_loop function. Each time a datum is received on input port 0, its first field must be a string. Unique strings are stored in an internal table, using the string as a "key". This is done via an STL "map" in the C++ version of count, and via a "dictionary" in the Python version of count. The value associated with each key is a count of the number of times the string has been received.

The count minnow shuts down when its input port is closed by receiving a sufficient number of "done" messages. This triggers the count minnow to send a series of datums to its ouput port 0, one for each unique word it has received. Each datum contains two fields. The first field is the count, the second is the string.

Data:

The count minnow must receive single field datums of type PHISH_STRING. It sends two-field datums of type (PHISH_INT32, PHISH_STRING).

Restrictions: none

Related minnows:

sort

echo program

Syntax:

```
echo
```

- this program takes no arguments

Examples:

```
wrapsink "echo"
wrapss "echo"
```

Description:

Echo is a stand-along non-PHISH program that can be wrapped with a PHISH minnow so it can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The echo program simply reads lines from stdin and echoes them to stdout. PHISH minnows that can wrap the echo program include the wrapsink and wrapss, which convert stdin/stdout into the receiving and sending of datums.

Ports:

The echo program does not call the PHISH library and thus does not use PHISH ports directly. But if it is wrapped with the wrapsink or wrapss minnows then they use one input port 0 to receive datums which are then read by the echo program via stdin. If it is wrapped with the wrapss minnow then it uses one output port 0 to send datums that are written to stdout by the echo program.

Operation:

The echo program simply reads a line of input from stdin and writes it to stdout. See the doc pages for the wrapsink or wrapss minnows for how they convert datums they receive to lines of text that the echo program can read from stdin, and how they convert lines of text that the echo program writes to stdout to datums they send.

Data:

The echo program does not call the PHISH library and thus does not deal directly with PHISH data types.

Restrictions:

The C++ version of the echo program allocates a buffer of size MAXLINE = 1024 bytes for reading a line from stdin. This can be changed (by editing minnow/echo.cpp) if longer lines are needed.

Related programs:

reverse

file2words minnow

Syntax:

```
file2words
```

- this minnow takes no arguments

Examples:

```
file2words
```

Description:

File2words is a PHISH minnow that can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows via its input and output ports.

The file2words minnow open a file, reads its contents, parses it into words separated by whitespace, and outputs each word.

Ports:

The file2words minnow uses one input port 0 to receive datums and one output port 0 to send datums.

Operation:

When it starts, the file2words minnow calls the phish_loop function. Each time a datum is received on input port 0, its first field is treated as a filename. The file is opened and its contents are read a line at a time. Each line is parsed into words, separated by whitespace. Each word is sent as an individual datum to its output port 0. The file is closed when it has all been read.

The filewords minnow shuts down when its input port is closed by receiving a sufficient number of "done" messages.

Data:

The file2words minnow msut receive single field datums of type PHISH_STRING. It also sends single field datums of type PHISH_STRING.

Restrictions:

The C++ version of the file2words minnow allocates a buffer of size MAXLINE = 1024 bytes for reading a line from a file. This can be changed (by editing minnow/file2words.cpp) if longer lines are needed.

It also assumes the filenames it receives are for text files, so that "whitespace" as defined in C or Python makes sense as a separator.

Related minnows:

filegen

filegen minnow

Syntax:

```
filegen path1 path2 ...
```

- path1,path2,... = one or more file or directory names

Examples:

```
filegen a1.txt a2.txt
filegen dir1 dir2 ... dir100
```

Description:

Filegen is a PHISH minnow that can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows via its input and output ports.

The filegen minnow generates a list of filenames from the filenames and directory names given to it as arguments. Each directory is opened (recursively) and scanned to generate filenames.

Ports:

The filegen minnow uses no input ports. It uses one output port 0 to send datums.

Operation:

When it starts, the filegen minnow loops over its input arguments. If the argument is a file, it sends the filename to its output port 0. If the argument is a directory name, it reads all the filenames in the directory and sends each one to its output port 0. If any entry in the directory is itself a directory, then it recurses and generates sends additional filenames to its output port 0.

When it has processed all its input arguments, the filegen minnow calls the phish_exit function to shut down.

Data:

Each datum the filegen minnow sends has a single field of type PHISH_STRING.

Restrictions: none

Related minnows:

file2words

phish_callback() function

C syntax:

```
void phish_callback(void (*alldonefunc)(), void (*abortfunc)(int*))
```

C examples:

```
#include "phish.h"
phish_callback(mydone,NULL);
phish_callback(NULL,myabort);
phish_callback(mydone,myabort);
```

C++ syntax:

```
void callback(void (*alldonefunc)(), void (*abortfunc)(int*))
```

C++ examples:

```
#include "phish.hpp"
phish::callback(mydone,NULL);
phish::callback(NULL,myabort);
phish::callback(mydone,myabort);
```

Python syntax:

```
def callback(alldonefunc, abortfunc)
```

Python examples:

```
import phish
phish.callback(mydone,None)
phish.callback(None,myabort)
phish.callback(mydone,myabort)
```

Description:

This is a PHISH library function which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

This function allows you to define 2 callback functions which the PHISH library will use to call back to the minnow under specific conditions. If they are not set, which is the NULL default, then the PHISH library does not make a callback.

The alldonefunc() function is used to specify a callback function invoked by the PHISH library when all the minnow's input ports have been closed. The callback function should have the following form:

```
void alldonefunc() { }
```

or

```
def alldonefunc()
```

in Python,

where "alldonefunc" is replaced by a function name of your choice. A minnow might use the function to print out some final statistics before the PHISH library exits. See the phish_close function and shutdown section of the Minnows doc page, for more discussion of how a school of minnows closes ports and shuts down.

The abortfunc() function is used to specify a callback function that invoked by the PHISH library when phish_abort is called, either by the minnow, or internally by the PHISH library.

The callback function should have the following form in C or C++:

```
void abortfunc(int* cancel) { }
```

or

```
def abortfunc(cancel)
```

in Python,

where "abortfunc" is replaced by a function name of your choice.

As explained on the phish_error doc page, the phish_abort() function may be called by the minnow directly, or implicitly by a call to phish_error(), and causes the minnow itself and the entire school of PHISH minnows to exit. If this callback is defined, the PHISH library will call the function before exiting. This can be useful if the minnow wishes to close files or otherwise clean-up. The function should not make additional calls to the PHISH library, as it may be in an invalid state, depending on the error condition.

The callback function may optionally set the "cancel" flag to a nonzero value to prevent the PHISH library from aborting the process.

Restrictions:

This function can be called anytime. It is the only PHISH library function that can be called before phish_init, which can be useful to perform needed clean-up via abortfunc() if phish_init() encounters an error.

Related commands:

phish_error, phish_abort

phish_check() function

C syntax:

```
void phish_check()
```

C examples:

```
phish_check();
```

C++ syntax:

```
void check()
```

C++ examples:

```
phish::check();
```

Python syntax:

```
def check()
```

Python examples:

```
import phish
phish.check()
```

Description:

This is a PHISH library function which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

This function is typically the final function called by a minnow during its setup phase, after the minnow has defined its input and output ports via the phish_input and phish_output functions. It must be called before any datums are received or sent to other minnows.

The function checks that the input and output ports defined by the minnow are consistent with their usage in the PHISH input script, as processed by the bait.py tool.

Specifically, it does the following:

- checks that required input ports are used by the script
- checks that no ports used by the script are undefined by the minnow
- opens all ports used by the script so that data exchanges can begin

---

Restrictions:

This function must be called after input and output ports have been defined, and before any datums are received or sent to other minnows.

Related commands:

phish_input, phish_output

phish_error() function

phish_warn() function

phish_abort() function

C syntax:

```
#include "phish.h"
void phish_error(char *str)
void phish_warn(char *str)
void phish_abort()
```

C examples:

phish_error("Bad datum received"); phish_warn("May overflow internal buffer"); phish_abort();

C++ syntax:

```
void phish::error(char *str)
void phish::warn(char *str)
void phish::abort()
```

C++ examples:

```
#include "phish.hpp"
phish::error("Bad datum received");
phish::warn("May overflow internal buffer");
phish::abort();
```

Python syntax:

```
def error(str)
def warn(str)
def abort()
```

Python examples:

```
import phish
phish.error("Bad datum received")
phish.warn("May overflow internal buffer")
phish.abort()
```

Description:

These are PHISH library functions which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

These functions print error or warning messages. The phish_error() and phish_abort() functions also cause a PHISH program and all of its minnows to exit.

These functions can be called by a minnow, but are also called internally by the PHISH library when error conditions are encountered.

Also note that unlike calling phish_exit, these functions do not close a minnow's input or output ports, or trigger "done" messages to be sent to downstream minnows. This means that no other minnows are explicitly told about the failed minnow. However, see the discussion below about the phish_abort() function and its effect on other minnows.

The phish_error() function prints the specified character string to the screen, then calls phish_abort().

The error message is printed with the following format:

```
PHISH ERROR: Minnow executable ID school-id # global-id: message
```

where "executable" is the name of executable minnow file (not the full path, just the filename), "school-id" is the ID of the minnow as specified in the PHISH input script, "global-id" is the global-ID of the minnow, and message is the error message. Each minnow has a global ID from 0 to Nglobal-1, where Nglobal is the total number of minnows in the net specified by the PHISH input script. This supplementary information is helpful in debugging which minnow generated the error message.

The phish_warn() function prints the specified character string to the screen, in the same format as phish_error(), execpt ERROR is replaced by WARNING. phish_abort() is not invoked and control is simply returned to the calling minnow which can continue executing.

The phish_abort() function invokes the user-specified abort callback function defined via phish_callback. If the callback function does not cancel the abort, the minnow exits.

For the MPI version of the PHISH library, phish_abort() invokes MPI_Abort(), which should force all minnows in the PHISH school to exit, along with the "mpirun" or "mpiexec" command that launched the net.

For the ZMQ version of the PHISH library, phish_abort() behavior is undefined. We intend that in future versions of PHISH, all minnows in the net will exit.

Restrictions: none

Related commands:

phish_exit

phish_query() function

C syntax:

```
int phish_query(char *keyword, int flag1, int flag2)
void phish_set(char *keyword, int flag1, int flag2)
```

- keywords for query = "idlocal" or "nlocal" or "idglobal" or "nglobal" or "inport/status" or "inport/nconnect" or "inport/nminnows" or "outport/status" or "outport/nconnect" or "output/nminnows" or "outport/direct"

```
idlocal
  flag1,flag2 = ignored
nlocal
  flag1,flag2 = ignored
idglobal
  flag1,flag2 = ignored
nglobal
  flag1,flag2 = ignored
inport/status
  flag1 = input port # (0 to Maxport-1)
  flag2 = ignored
inport/nconnect
  flag1 = input port # (0 to Maxport-1)
  flag2 = ignored
inport/nminnow
  flag1 = input port # (0 to Maxport-1)
  flag2 = connection # on that port (0 to Nconnect-1)
outport/status
  flag1 = output port # (0 to Maxport-1)
  flag2 = ignored
outport/nconnect
  flag1 = output port # (0 to Maxport-1)
  flag2 = ignored
outport/nminnow
  flag1 = output port # (0 to Maxport-1)
  flag2 = connection # on that port (0 to Nconnect-1)
outport/direct
  flag1 = output port # (0 to Maxport-1)
  flag2 = ignored
```

- keywords for set = "ring/receiver"

```
ring/receiver
  flag1 = input port # (0 to Maxport-1)
  flag2 = receiver ID (0 to Nring-1)
```

C examples:

```
#include "phish.h"
int nlocal = phish_query("nlocal",0,0);
int nrecv = phish_query("outport/direct",2,0);
phish_set("ring/receiver",0,3);
```

C++ syntax:

```
int query(char *keyword, int flag1, int flag2)
```

```
void set(char *keyword, int flag1, int flag2)
```

C++ examples:

```
#include "phish.hpp"
int nlocal = phish::query("nlocal",0,0);
int nrecv = phish::query("outport/direct",2,0);
phish::set("ring/receiver",0,3);
```

Python syntax:

```
def query(str,flag1,flag2)
def set(str,flag1,flag2)
```

Python examples:

```
import phish
nlocal = phish.query("nlocal",0,0)
nrecv = phish.query("outport/direct",2,0)
phish.set("ring/receiver",0,3)
```

Description:

These are PHISH library functions which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

These functions are used to query and reset information stored internally in PHISH. New keywords may be added as usage cases arise.

For phish_query, the "idlocal", "nlocal", "idglobal", and "nglobal" keywords return info about the minnow and its relation to other minnows running the PHISH program. These keywords ignore the flag1 and flag2 values; they can simply be set to 0.

A PHISH program typically includes one or more sets of minnows, as specified in a PHISH input script. Each minnow in each set is an individual process. In a local sense, each minnow has a local-ID from 0 to Nlocal-1 within its set, where *Nlocal* is the number of minnows in the set. Globally, each minnow has a global-ID from 0 to Nglobal-1, where *Nglobal* is the total number of minnows. The global-IDs are ordered by set, so that minnows within each set have consecutive IDs. These IDs enable the PHISH library to orchestrate communication of datums between minnows in different sets. E.g. when running the MPI version of the PHISH library, the global-ID corresponsds to the rank ID of an MPI process, used in MPI_Send() and MPI_Recv() function calls.

For phish_query, the "inport/status", "inport/nconnect", and "inport/nminnows" keywords return info about the input ports that connect to the minnow by which it receives datums from other minnows. Likewise, the "outport/status", "outport/nconnect", "output/nminnows", and "output/direct" keywords return info about the output ports the minnow connects to by which it sends datums to other minnows.

All of these keywords require the use of *flag1* to specify the input or output port, which is a number from 0 to Maxport-1. Some of them, as noted below, require the use of *flag2* to specify the connection #, which is a number from 0 to Nconnect-1.

See this section of the PHISH Minnows doc page for more information about input and output ports.

See the hook command which is processed by the bait.py tool in a PHISH input script, to establish connections between sets of minnows.

The "status" keyword returns the status of the port, which is one of the following values:

- unused = 0
- open = 1
- closed = 2

The "nconnect" keyword returns the number of sets of minnows that are connected to a port.

The "nminnows" keyword returns the number of minnows connected to a port thru a specific connection, as specified by flag2.

The "outport/direct" keyword returns the number of minnows connected to an output port thru a connection of style *direct*. The first such connection found is used to return this value, so if another *direct* connection is desired, the "outport/nminnows" keyword should be used.

See the phish_send_direct function for a discussion of how datums are sent via *direct* style connections, and why this particular phish_query() keyword can be useful.

For phish_set, the "ring/receiver" keyword changes the minnow that this minnow sends messages to. This keyword can only be used when the minnow is part of school of minnows that is exchanging datums via a "ring" connection; see the hook command in PHISH input scripts that defines the ring connection. This keyword can be used to effectively permute the ordering of the minnows in the ring.

For ring/receiver, *flag1* is the output port number. *Flag2* is the new receiving minnow to send datums to on that port. It should be a value from 0 to Nring-1 inclusive, where Nring = the # of minnows in the ring.

Restrictions: none

Related commands:

phish_init

phish_init() function

C syntax:

```
int phish_init(int *argc, char ***argv)
```

C examples:

```
phish_init(&argc,&argv);
```

C++ syntax:

```
void init(int& argc, char*& argv)
```

C++ examples:

```
phish::init(argc, argv);
```

Python syntax:

```
def init(argv)
```

Python examples:

```
import phish
argv = phish.init(sys.argv)
```

Description:

This is a PHISH library function which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

A PHISH program typically includes one or more schools of minnows, as specified in a PHISH input script. Each minnow in each school is an individual process. Locally, each minnow has a local-ID from 0 to Nlocal-1 within its school, where *Nlocal* is the number of minnows in the school. Globally, each minnow has a global-ID from 0 to Nglobal-1, where *Nglobal* is the total number of minnows in the net. The global-IDs are ordered by school, so that minnows within each school have consecutive IDs. These IDs enable the PHISH library to orchestrate communication of datums between minnows in different schools. E.g. when running the MPI version of the PHISH library, the global-ID corresponsds to the rank ID of an MPI process, used in MPI_Send() and MPI_Recv() function calls.

See the phish_query function for how a minnow can find out these values from the PHISH library.

The phish_init() function must be the first call to the PHISH library made by a minnow. Since it alters the command-line arguments passed to the minnow, it is typically the first executable line of a minnow program.

It's purpose is to initialize the library using special command-line arguments passed to the minnow when it was launched, typically by the the bait.py tool that parses a PHISH input script.

The two arguments to phish_init() are pointers to the number of command-line arguments, and a pointer to the arguments themselves as an array of strings. These are passed as pointers, because the PHISH library reads and removes the PHISH-specific arguments. It then returns the remaining minnow-specific arguments, which the minnow can read and process. As is standard with C programs, the returned value of argv includes argv[0] which is the name of the minnow executable.

Note that in the Python version of phish.init(), the full argument list is passed as an argument, and the truncated argument list is returned.

Following are some sample switches and arguments that the MPI version of the PHISH library looks for and processes. These are generated automatically by the bait.py tool when it processes a PHISH input script so you don't need to think about these arguments, but it may be helpful in understanding how PHISH works. Note that these arguments are normally invisible to the user; their format and number may be changed in future versions of PHISH.

- --phish-backend mpi
- --phish-minnow ID Nlocal Nprev
- --phish-memory N
- --phish-in sprocs sfirst sport style rprocs rfirst rport
- --phish-out sprocs sfirst sport style rprocs rfirst rport

The *--phish-backend* switch appears once, and is followed by the version of the PHISH library specified by bait.py. This allows the PHISH library to detect incompatible runtime environments, e.g. mixing minnows linked against the socket version of the PHISH library with minnows using the MPI version.

The *--phish-minnow* switch appears once. ID is the school ID in the PHISH input script. The *Nlocal* argument was explained above. *Nprev* is the total number of minnows in sets of minnows previous to this one. It is used to infer the *local-ID* value discussed above.

The *--phish-memory* switch changes a default setting within the PHISH library. There is a similar command for each keyword supported by the bait.py set command.

The *--phish-memory* value *N* sets the maximum size of the buffers used to send and receive datums. See the set command of the bait.py tool for more information on the settings of this switch.

The *--phish-in* switch appears once for every connection the minnow has with other minnows, where it is a receiver of datums. See the hook command in PHISH input scripts processed by the bait.py tool, for more information.

Sprocs, *sfirst*, and *sport* refer to the set of minnows sending to this minnow. They are respectively, the number of minnows in the set, the global ID of the first minnow in the set, and the output port used by those minnows. *Rprocs*, *rfirst*, and *rport* refer to the set of minnows receivng the datums, i.e. the set of minnows this minnow belongs to. They are respectively, the number of minnows in the set, the global ID of the first minnow in the set, and the input port used by those minnows. *Style* is the connection style, as specified by the hook command in the PHISH input script processed by the bait.py tool. E.g. *style* is a word like "single" or "hashed". If it is "subscribe", then extra info about the external host and its TCP port is appended to the *style*, e.g. "subscribe/www.foo.com:25".

The *--phish-out* switch appears once for every connection the minnow has with other minnows, where it is a sender of datums. See the hook command in PHISH input scripts processed by the bait.py tool, for more information.

Sprocs, *sfirst*, and *sport* refer to the set of minnows sending datums, i.e. the set of minnows this minnow belongs to. They are respectively, the number of minnows in the set, the global ID of the first minnow in the set, and the output port used by those minnows. *Rprocs*, *rfirst*, and *rport* refer to the set of minnows receivng the datums. They are respectively, the number of minnows in the set, the global ID of the first minnow in the set, and the input port used by those minnows. *Style* is the connection style, as specified by the hook command in the PHISH input script processed by the bait.py tool. E.g. *style* is a word like "single" or "hashed". If it is "publish", then extra info about the TCP port is appended to the *style*, e.g. "publish/25".

The PHISH library ignores any remaining arguments, returning them to to the minnow caller, including the name of the minnow executable in argv[0].

The phish_init() function also flags each specified input port and output port with a CLOSED status, instead of UNUSED. See the hook command for the bait.py tool for more info about communication ports. See the phish_input and phish_output functions for more info about port status.

The "C" binding to phish_init() returns a nonzero value if there were errors, otherwise zero. The "C++" binding to phish::init() throws an exception if there were errors.

Restrictions: none

Related commands:

phish_query

phish_repack

phish_pack_raw

phish_pack_char

phish_pack_int8

phish_pack_int16

phish_pack_int32

phish_pack_int64

phish_pack_uint8

phish_pack_uint16

phish_pack_uint32

phish_pack_uint64

phish_pack_float

phish_pack_double

phish_pack_string

phish_pack_int8_array

phish_pack_int16_array

phish_pack_int32_array

phish_pack_int64_array

phish_pack_uint8_array

phish_pack_uint16_array

phish_pack_uint32_array

phish_pack_uint64_array

phish_pack_float_array

phish_pack_double_array

phish_pack_pickle

C syntax:

```
void phish_repack();
void phish_pack_raw(char *buf, int32_t n);
void phish_pack_char(char value);
void phish_pack_int8(int8_t value);
void phish_pack_int16(int16_t value);
void phish_pack_int32(int32_t value);
void phish_pack_int64(int64_t value);
void phish_pack_uint8(uint8_t value);
void phish_pack_uint16(uint16_t value);
void phish_pack_uint32(uint32_t value);
void phish_pack_uint64(uint64_t value);
void phish_pack_float(float value);
void phish_pack_double(double value);
void phish_pack_string(char *str);
void phish_pack_int8_array(int8_t *vec, int32_t n);
void phish_pack_int16_array(int16_t *vec, int32_t n);
void phish_pack_int32_array(int32_t *vec, int32_t n);
void phish_pack_int64_array(int64_t *vec, int32_t n);
void phish_pack_int8_array(int8_t *vec, int32_t n);
void phish_pack_int16_array(int16_t *vec, int32_t n);
void phish_pack_int32_array(int32_t *vec, int32_t n);
void phish_pack_int64_array(int64_t *vec, int32_t n);
void phish_pack_float_array(float *vec, int32_t n);
void phish_pack_double_array(double *vec, int32_t n);
void phish_pack_pickle(char *buf, int32_t n);
```

C examples:

```
#include "phish.h"
int n;
uint64_t nlarge;
phish_repack();
phish_pack_char('a');
phish_pack_int32(n);
phish_pack_uint64(nlarge);
phish_pack_string("this is my data");
phish_pack_double_array(vec,n);
```

C++ syntax:

```
void repack();
void pack(const char *buf, int32_t n);
void pack(char value);
void pack(int8_t value);
void pack(int16_t value);
void pack(int32_t value);
void pack(int64_t value);
void pack(uint8_t value);
void pack(uint16_t value);
```

```
void pack(uint32_t value);
void pack(uint64_t value);
void pack(float value);
void pack(double value);
void pack(const char *str);
void pack(const std::string& str);
void pack(int8_t *vec, int32_t n);
void pack(int16_t *vec, int32_t n);
void pack(int32_t *vec, int32_t n);
void pack(int64_t *vec, int32_t n);
void pack(int8_t *vec, int32_t n);
void pack(int16_t *vec, int32_t n);
void pack(int32_t *vec, int32_t n);
void pack(int64_t *vec, int32_t n);
void pack(float *vec, int32_t n);
void pack(double *vec, int32_t n);
```

C++ examples:

```
#include "phish.hpp"
int n;
uint64_t nlarge;
phish::repack();
phish::pack('a');
phish::pack(n);
phish::pack(nlarge);
phish::pack("this is my data");
phish::pack(vec,n);
```

Python syntax:

```
def repack()
def pack_raw(buf,n)
def pack_char(value)
def pack_int8(value)
def pack_int16(value)
def pack_int32(value)
def pack_int64(value)
def pack_uint8(value)
def pack_uint16(value)
def pack_uint32(value)
def pack_uint64(value)
def pack_float(value)
def pack_double(value)
def pack_string(str)
def pack_int8_array(vec)
def pack_int16_array(vec)
def pack_int32_array(vec)
def pack_int64_array(vec)
def pack_int8_array(vec)
def pack_int16_array(vec)
def pack_int32_array(vec)
def pack_int64_array(vec)
def pack_float_array(vec)
def pack_double_array(vec)
def pack_pickle(obj)
```

Python examples:

```
import phish
phish.repack()
phish.pack_char('a')
```

```
phish.pack_int32(n)
phish.pack_uint64(nlarge)
phish.pack_string("this is my data")
phish.pack_double_array(vec)
phish.pack_int32_array(1,10,20,4)
phish.pack_pickle(59899.984)
phish.pack_pickle(1,10,20,4)
foo1 = 1,2,3,"flag",7.0,10.0
phish.pack_pickle(foo1)
foo2 = "key1" : "value1", "dog" : "cat"
phish.pack_pickle(foo2)
```

Description:

These are PHISH library functions which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

These functions are used to pack individual values into a datum as fields before sending the datum to another minnow.

As discussed in this section of the PHISH Library doc page, datums sent and recived by the PHISH library contain one or more fields. A field is a fundamental data type, such as a "32-bit integer" or "vector of doubles" or a NULL-terminated character string. Except for phish_repack, these pack functions add a single field to a datum by packing the data into a buffer, using integer flags to indicate what type and length of data comes next. Unpack functions allow the minnow to extract data from the datum, one field at a time.

Once data has been packed, the minnow may re-use the variables that store the data; the pack functions copy the data into an internal send buffer inside the PHISH library.

The repack() function packs all the fields of the most recently received datum for sending. This is a mechanism for sending an entire dataum as-is to another minnow.

The repack() function can be used in conjunction with other pack functions. E.g. pack functions can be used before or after the repack() function to prepend or append additional fields to a received datum.

The various pack functions correspond one-to-one with the kinds of fundamental data that can be packed into a PHISH datum:

- phish_pack_raw() = pack a string of raw bytes of length *n*
- phish_pack_char() = pack a single character
- phish_pack_int*() = pack a single int of various sizes (8,16,32,64 bits)
- phish_pack_uint*() = pack a single unsigned int of various sizes (8,16,32,64 bits)
- phish_pack_float() = pack a single double
- phish_pack_double() = pack a single double
- phish_pack_string() = pack a C-style NULL-terminated string of bytes, including the NULL
- phish_pack_int*_array() = pack *n* int values from *vec*
- phish_pack_uint*_array() = pack *n* uint64 values from *vec*
- phish_pack_float_array() = pack *n* float values from *vec*
- phish_pack_double_array() = pack *n* double values from *vec*

Note that for the array functions, *n* is typed as an int32_t which is a 32-bit integer. In C or C++, the minnow can simply declare *n* to be an "int" and any needed casting will be performed automatically. The only case where this will fail (with a compile-time error) is if the native "int" on a machine is a 64-bit int.

Phish_pack_raw() can be used with whatever string of raw bytes the minnow puts into its own buffer, pointed to by the *buf* argument, e.g. a C data structure containing a collection of various C primitive data types. The "int*" data type refers to signed integers of various lengths. The "uint*" data type refers to unsigned integers of various lengths. Phish_pack_string() will pack a standard C-style NULL-terminated string of bytes and include the NULL. The array pack functions expect a *vec* pointer to point to a contiguous vector of "int*" or "uint*" or floating point values.

Note that the Python interface to the pack functions is slightly different than the C or C++ interface.

The array pack functions do not take a length argument *n*. This is because Python can query the length of the vector itself.

The pack_pickle() function is unique to Python, it should not normally called from C or C++. It will take any Python object as an argument, a fundamental data type like an integer or floating-point value or sting, or a more complex Python object like a list, or dictionary, or list of arbitrary objects. Python converts the object into a string of bytes via its "pickling" capability, before it is packed into the PHISH library send buffer. When that field in the datum is unpacked, via a call to the phish_unpack function, the bytes are "unpickled" and the Python object is recreated with its internal structure intact. Thus minnows written in Python can exchange Python objects transparenlty.

Restrictions: none

Related commands:

phish_send, phish_unpack

phish_input() function

phish_output() function

C syntax:

```
void phish_input(int iport, void (*datumfunc)(int), void (*donefunc)(), reqflag)
void phish_output(int iport)
```

C examples:

```
#include "phish.h"
phish_input(0,count,NULL,1);
phish_input(1,count,mydone,0);
phish_output(0);
```

C++ syntax:

```
void input(int port, void (*datumfunc)(int), void (*donefunc)(), bool required=true)
void output(int port)
```

C++ examples:

```
#include "phish.h.pp"
phish::input(0,count,NULL,true);
phish::input(1,count,mydone,false);
phish::output(0);
```

Python syntax:

```
def input(iport,datumfunc,donefunc,reqflag)
def output(iport)
```

Python examples:

```
import phish
phish.input(0,count,None,1)
phish.input(1,count,mydone,0)
phish.output(0)
```

Description:

These are PHISH library functions which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

The phish_input() and phish_output() functions define input and output ports for the minnow. An input port is where datums are sent by other minnows, so they can be read by this minnow. An output port is where the minnow sends datums to route them to the input ports of other minnows. These inter-minnow connections are setup by the hook command in a PHISH input script, as discussed on the bait.py doc page.

A minnnow can define and use multiple input and output ports, to send and receive datums of different kinds to different sets of minnows. Both input and output ports are numbered from 0 to Pmax-1, where Pmax = the maximum allowed ports, which is a hard-coded value for MAXPORT in src/phish.cpp. It is currently set to 16;

most minnows use 1 or 2. Note that a single port can be used to send or receive datums to many other minnows (processors), depending on the connection style. See the hook command for details.

The minnow should make one call to phish_input() for each input port it uses, whether or not a particular PHISH input script actually connects to the port. Specify *reqflag* = 1 if a PHISH input script must specify a connection to the input port in order to use the minnow; specify *reqflag* = 0 if it is optional. The phish_check function will check for compatibility between the PHISH input script and the minnow ports.

Two callback function pointers are passed as arguments to phish_input(). Either or both can be specied as *NULL*, or *None* in the Python version, if the minnow does not require a callback. Note that multiple input ports can use the same callback functions.

The first callback is *datumfunc*, and is called by the PHISH library each time a datum is received on that input port.

The *datumfunc* function should have the following form:

```
void datumfunc(int nfields) { }
```

or

```
def datumfunc(nfields)
```

in Python,

where "datumfunc" is replaced by a function name of your choice. The function is passed "nfields" = the # of fields in the received datum. See the phish_unpack and phish_datum doc pages for info on how the received datum can be further processed.

The second callback is *donefunc*, and is a called by the PHISH library when the input port is closed.

The *donefunc* function should have the following form:

```
void donefunc() { }
```

or

```
def donefunc()
```

in Python,

where "donefunc" is replaced by a function name of your choice. A minnow might use the function to print out some statistics about data received thru that input port, or its closure might trigger further data to be sent downstream to other minnows. See the phish_close function and shutdown section of the Minnows doc page, for more discussion of how a school of minnows closes ports and shuts down.

The minnow should make one call to phish_output() for each output port it uses, whether or not a particular PHISH input script actually connects to the port. Usage of an output port by an input script is always optional. This makes it easy to develop and debug a sequence of pipelined operations, one minnow at a time, without requiring a minnow's output to be used by an input script.

Restrictions:

These functions cannot be called after phish_check has been called.

Related commands:

phish_check, phish_close

phish_queue() function

phish_dequeue() function

phish_nqueue() function

C syntax:

```
int phish_queue()
```

```
int phish_dequeue(int n)
```

```
int phish_nqueue()
```

C examples:

```
nq = phish_queue();
nvalues = phish_dequeue(0);
nq = phish_nqueue();
```

Python syntax:

```
def queue()
```

```
def dequeue(n)
```

```
def nqueue()
```

Python examples:

```
import phish
nq = phish.queue()
nvalues = phish.dequeue(0)
nq = phish.nqueue()
```

Description:

These are PHISH library functions which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

These functions are used to store and retrieve datums in an internal queue maintained by the PHISH library. This can be useful if a minnow receives a datum but wishes to process it later.

The phish_queue() function stores the most recently received datum in the internal queue. It returns the number of datums in the queue, which includes the one just stored.

The phish_queue() function does not conflict with phish_unpack or phish_datum functions. They can be called before or after a phish_queue() call.

The phish_dequeue() function retrieves a stored datum from the internal queue and copies it into the receive buffer, as if it had just been received. The datum is deleted from the queue, though it can be requeued via a subsequent call to phish_queue.

After a call to phish_dequeue, the datum can be unpacked or its attributes queried via the phish_unpack or phish_datum functions, as if it just been received.

The input parameter "n" for phish_dequeue is the index of the datum to retrieve. N can be any value from 0 to Nqueue-1 inclusive, where Nqueue is the number of datums in the queue. Thus you can easily retrieve the oldest or newest datum in the queue.

The phish_nqueue() function returns the number of datums currently held in the internal queue.

Restrictions: none

Related commands:

phish_recv, phish_datum

phish_loop() function

phish_probe() function

phish_recv() function

C syntax:

```
void phish_loop()
void phish_probe(void (*probefunc)())
int phish_recv()
```

C examples:

```
#include "phish.h"
phish_loop();
phish_probe(count);
int n = phish_recv();
```

C++ syntax:

```
void loop()
void probe(void (*probefunc)())
int recv()
```

C++ examples:

```
#include "phish.hpp"
phish::loop();
phish::probe(count);
int n = phish::recv();
```

Python syntax:

```
def loop()
def probe(probefunc)
def recv()
```

Python examples:

```
import phish
phish.loop()
phish.probe(count)
n = phish.recv()
```

Description:

These are PHISH library functions which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

These functions are used to receive datums sent by other minnows.

All received datums arrive on input ports the minnow defines and which the PHISH input script uses to route datums from one set of minnows to another set.

The functions documented on this page receive the next datum, whichever input port it arrives on. It is up to the minnow to take the appropriate port-specific action if necessary. This can be done by defining a port-specific callback function via the phish_input function. Or by querying what port the datum was received on via the phish_datum function.

Note that datums sent by a minnow to itself are always processed first by all of these functions. For datums sent from another minnow, they are processed in the order they are received, i.e. first-come, first-served.

The phish_loop() function turns control over to the PHISH library. It will wait for the next datum to arrive on any input port. When it does one of three things happen:

(1) For a regular datum, phish_loop() will make a callback to the minnow, to the *datum* callback function assigned to the input port the datum was received on. See the phish_input function for how this callback function is assigned. When the callback function returns, control is returned to phish_loop().

(2) For a datum that signals the closure of an input port, phish_loop() will make a callback to the minnow, to the *done* callback function assigned to the input port the datum was received on. See the phish_input function for how this callback function is assigned. When the callback function returns, control is returned to phish_loop().

(3) For a datum that closes the last open input port, step (2) is performed, and then an additional callback to the minnow is made, to the *alldone* callback function (optionally) assigned by the phish_callback function. When the callback function returns, control is returned to phish_loop().

After option (3) has occurred, phish_loop() returns, giving control back to the minnow. Typically, the minnow will then clean up and call phish_exit, since all its input ports are closed and no more datums can be received.

The phish_probe() function is identical to phish_loop(), except that instead of waiting for the next datum to arrive, phish_probe() checks if a datum has arrived. If not, then it immediately calls the specified *probefunc* callback function. This allows the minnow to do useful work while waiting for the next datum to arrive.

The *probefunc* function should have the following form:

```
void probefunc() { }
```

or

```
def probefunc()
```

in Python,

where "probefunc" is replaced by a function name of your choice. When the *probefunc* callback function returns, control is returned to phish_probe().

Note that just like phish_loop(), phish_probe() will not return control to the minnow, until option (3) above has occured, i.e. all input ports have been closed.

The phish_recv() function allows the minnow to request datums explicitly, rather than be handing control to phish_loop() or phish_probe() and being called back to by those functions.

The phish_recv() function checks if a datum has arrived and returns regardless. It returns a value of 0 if no datum is available. It returns a value N > 0 if a datum has arrived, with N = the number of fields in the datum. See the phish_unpack and phish_datum doc pages for info on how the received datum can be further processed.

If a datum is received that signals the closure of an input port, then phish_recv() will perform the same options (2) and (3) listed above, making callbacks to the *done* callback function and *alldone* callback function as appopriate, and then return with a value of -1.

Restrictions:

These functions can only be called after phish_check has been called.

Related commands:

phish_input, phish_done

phish_send() function

phish_send_key() function

phish_send_direct() function

C syntax:

```
void phish_send(int iport)
void phish_send_key(int iport, char *key, int nbytes)
void phish_send_direct(int iport, int receiver)
```

C examples:

```
#include "phish.h"
phish_send(0);
phish_send_key(1,id,strlen(id));
phish_send_direct(0,3);
```

C++ syntax:

```
void send(int iport)
void send_key(int iport, char *key, int nbytes)
void send_direct(int iport, int receiver)
```

C++ examples:

```
#include "phish.hpp"
phish::send(0);
phish::send_key(1,id,strlen(id));
phish::send_direct(0,3);
```

Python syntax:

```
def send(iport)
def send_key(iport,key)
def send_direct(iport,receiver)
```

Python examples:

```
import phish
phish.send(0)
phish.send_key(1,id)
phish.send_direct(0,3)
```

Description:

These are PHISH library functions which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

These functions are used to send datums to other minnows. Before a datum can be sent, it must be packed into a buffer. See the doc page for the phish_pack functions to see how this is done.

All datums are sent via output ports the minnow defines and which the PHISH input script uses to route datums from one set of minnows to another set. Thus these send functions all take an *iport* argument to specify which output port to send thru.

The specific minnow(s) that the datum will be sent to is determined by the connection style(s) defined for the output port. See the PHISH input script hook command, as discussed on the bait.py tool doc page, for details. Some connection styles require additional information from the minnow to route the datum to the desired minnow. This is the reason for the phish_send_key() and phish_send_direct() variants of phish_send().

The phish_send() function sends a datum to the specified *iport* output port.

This generic form of a send can be used for all connection styles except the *hashed* and *direct* styles. See the PHISH input script hook command for details. Note that multiple sets of receiving minnows, each with their own connection style, can be connected to the same output port.

If phish_send() is used with a *hashed* or *direct* connection style, an error will result.

The phish_send_key() function sends a datum to the specified *iport* output port and allows specification of a byte string or *key* of length *nbytes*, which will be *hashed* by the PHISH library and converted into an index for choosing a specific receiving processor to send the datum to.

This form of sending must be used for a *hashed* connection style. See the PHISH input script hook command for details. If a connection style to a specific output port is not a *hashed* style, then the *key* and *nbytes* arguments are ignored, and the generic phish_send() form is used to send the datum.

The phish_send_direct() function sends a datum to the specified *iport* output port and allows a specific receiving minnow to be selected via the *receiver* argument. The *receiver* is an integer offset into the set of receiving minnows connected to this output port. If there are *M* minnows in the receiving set, then 0 <= *receiver* < M is required. The phish_query function can be used to query information about the receiving set of minnows. For example this phish_query() call would return M, assuming the receiving processors are connected to output port 0.

```
int m = phish_query("outport/direct",0,0);
```

This form of sending must be used for a *direct* connection style. See the PHISH input script hook command for details. If one or more of the connection styles connected to the output port is not a *direct* style, then the *reciever* argument is ignored, and the generic phish_send() form is used to send the datum.

Restrictions: none

Related commands:

phish_pack

phish_exit() function

phish_close() function

C syntax:

```
void phish_exit()
void phish_close(int iport)
```

C examples:

```
#include "phish.h"
phish_exit();
phish_close(0);
```

C++ syntax:

```
void exit()
void close(int iport)
```

C++ examples:

```
#include "phish.hpp"
phish::exit();
phish::close(0);
```

Python syntax:

```
def exit()
def close(iport)
```

Python examples:

```
import phish
phish.exit();
phish.close(0);
```

Description:

These are PHISH library functions which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

These functions serve to shutdown a running minnow, either entirely or a portion of its output capabilities. They trigger the closing of a minnow's output port(s) which notifies downstream minnows, so they also can clean-up and exit.

See this section of the Minnows doc page for a discussion of shutdown options for PHISH programs.

The phish_exit() function is the most commonly used mechanism for performing an orderly shutdown of a PHISH program. Once called, no further calls to the PHISH library can be made by a minnow, so it is often the final line of a minnow program.

When phish_exit() is called it performs the following operations:

- print stats about the # of datums received and sent by the minnow
- warn if any input port is not closed
- close all output ports
- free internal memory allocated by the PHISH library
- shutdown communication protocols to other minnows

The stats message is printed with the same supplementary information as the phish_error function, to identify the minnow that printed it.

Closing a minnow's output port involves sending a "done" message to each minnow (in each set of minnows) connected as a receiver to that port, so that they know to expect no more datums from this minnow.

When all the minnows in a set have invoked phish_exit() to close an output port, each downstream minnow that receives output from this port will have received a series of "done" messages on its corresponding input port. Each minnow keeps a count of the total # of minnows that send to that port, so it will know when the requisite number of done messages have been received to close the input port.

In the MPI version of the library, the final step is to invoke MPI_Finalize(), which means no further MPI calls can be made by the minnow.

In the ZMQ version of the library, the final step is to close any open ZMQ context(s), so no further ZMQ calls can be made by the minnow.

Note that this function is often called directly by the most upstream minnow(s) in a PHISH school, when they are done with their task (e.g. reading data from a file).

Other downstream minnows often call phish_exit() after the phish_loop or phish_probe function returns control to the minnow, since that only occurs when all the minnow's input ports have been closed. In this manner, the shutdown procedure cascades from minnow to minnow.

The phish_close() function is used less often than the phish_exit() function. It can be useful when some minnow in the middle of a data processing pipeline needs to trigger an orderly shutdown of the PHISH program.

Phish_close() closes the specified *iport* output port of a minnow. This procedure involves sending a "done" message to each minnow (in each set of minnows) connected as a receiver to that port, so that they know to expect no more datums from this minnow.

When all the minnows in a set have invoked phish_close() on an output port, each downstream minnow that receives output from this port will have received a series of "done" messages on its corresponding input port. Each minnow keeps a count of the total # of minnows that send to that port, so it will know when the requisite number of done messages have been received to close the input port. As input ports are closed, this typically triggers the minnow to invoke phish_exit() or phish_close(). In this manner, the shutdown procedure cascades from minnow to minnow.

This function does nothing if the specified output port is already closed.

Restrictions: none

Related commands:

phish_loop, phish_probe

phish_timer() function

C syntax:

```
double phish_timer()
```

C examples:

```
#include "phish.h"
double t1 = phish_timer();
...
double t2 = phish_timer();
printf("Elapsed time = %g\n",t2-t1);
```

C++ syntax:

```
double timer()
```

C++ examples:

```
#include "phish.hpp"
double t1 = phish::timer();
...
double t2 = phish::timer();
printf("Elapsed time = %g\n",t2-t1);
```

Python syntax:

```
def timer()
```

Python examples:

```
import phish
t1 = phish.timer();
...
t2 = phish.timer();
print "Elapsed time =",t2-t1
```

Description:

This is a PHISH library function which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

This function provides a portable means to time operations within a minnow. The function returns the current wallclock time as a timestamp measured in seconds. To calculate an elapsed time, you need to bracket a section of code with 2 calls to phish_timer() and compute the difference between the 2 returned times, as in the example above.

Restrictions: none

Related commands: none

phish_unpack() function

phish_datum() function

C syntax:

```
int phish_unpack(char **buf, int32_t *len)
int phish_datum(int flag)
```

C examples:

```
#include "phish.h"
char *buf;
int len;
int type = phish_unpack(&buf,&len);
int iport = phish_datum(1);
```

C++ syntax:

```
int unpack(char **buf, int32_t *len)
int datum(int flag)
```

C++ examples:

```
#include "phish.hpp"
char *buf;
int len;
int type = phish::unpack(&buf,&len);
int iport = phish::datum(1);
```

Python syntax:

```
def unpack()
def datum(flag)
```

Python examples:

```
import phish
type,value,len = phish.unpack()
iport = phish.datum(1)
```

Description:

These are PHISH library functions which can be called from a minnow application. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library.

These functions are used to unpack a datum after it has been received from another minnow or query other info about the datum.

As discussed in this section of the PHISH Library doc page, datums sent and recived by the PHISH library contain one or more fields. A field is a fundamental data type, such as an "32-bit integer" or "vector of doubles" or a NULL-terminated character string. These fields are packed into a contiguous byte string when then are sent, using integer flags to indicate what type and length of data comes next. These unpack functions allow the minnow to extract data from the datum, one field at a time.

Note that these functions return pointers to the internal buffer holding the datum within the PHISH library. The buffer will be overwritten when the minnow returns control to the PHISH library and the next datum is received. Typically this occurs when a callback function in the minnow returns. This means that if you want the data to persist within the minnow, you must make a copy. It is OK to unpack several fields from the same datum before making copies of the fields. It is also OK to pack one or more received fields for sending and wait to send it until after another datum is received. This is because calls to "phish_pack" functions copy data into a separate send buffer.

The phish_unpack() function returns the next field and its length, from the most recently received datum. Note that *len* is typed as a pointer to int32_t which is a 32-bit integer. In C or C++, the minnow can simply declare *len* to be a pointer to "int" and the function will work as expected. The only case where this will fail (with a compile-time error) is if the native "int" on a machine is not a 32-bit int.

Phish_unpack returns an integer flag set to one of these values (defined in src/phish.h):

- PHISH_CHAR = 0
- PHISH_INT8 = 1
- PHISH_INT16 = 2
- PHISH_INT32 = 3
- PHISH_INT64 = 4
- PHISH_UINT8 = 5
- PHISH_UINT16 = 6
- PHISH_UINT32 = 7
- PHISH_UINT64 = 8
- PHISH_FLOAT = 9
- PHISH_DOUBLE = 10
- PHISH_RAW = 11
- PHISH_STRING = 12
- PHISH_INT8_ARRAY = 13
- PHISH_INT16_ARRAY = 14
- PHISH_INT32_ARRAY = 15
- PHISH_INT64_ARRAY = 16
- PHISH_UINT8_ARRAY = 17
- PHISH_UINT16_ARRAY = 18
- PHISH_UINT32_ARRAY = 19
- PHISH_UINT64_ARRAY = 20
- PHISH_FLOAT_ARRAY = 21
- PHISH_DOUBLE_ARRAY = 22
- PHISH_PICKLE = 23

PHISH_CHAR, PHISH_INT*, PHISH_UINT*, PHISH_FLOAT, and PHISH_DOUBLE are a single character, a signed integer (of length 8,16,32,64 bits), an unsigned integer (of length 8,16,32,64 bits), a float (typically 4 bytes), and a double (typically 8 bytes).

PHISH_RAW is a string of raw bytes which can store whatever the sending minnow put into its send buffer, e.g. a C data structure containing a collection of various C primitive data types.

PHISH_RAW is a string of raw bytes which minnows can format in any manner, e.g. a C data structure containing a collection of various C primitive data types. PHISH_STRING is a standard C-style NULL-terminated C-string. The NULL is included in the field.

The ARRAY types are contiguous sequences of int*, uint*, float, or double values, packed one after the other.

PHISH_PICKLE is an option available when using the Python wrapper on the PHISH library to encode arbitrary Python objects in pickled form as a string of bytes. It should not normally be used in a minnow written in C or C++.

Phish_unpack also returns *buf* and *len*. *Buf* is a char pointer to where the field starts. You will need to cast this to the appropriate data type before accessing the data if it is not a character string. *Len* is the length of the field, with the following meanings:

- PHISH_CHAR: len = 1
- PHISH_INT*: len = 1
- PHISH_UINT*: len = 1
- PHISH_FLOAT: len = 1
- PHISH_DOUBLE: len = 1
- PHISH_RAW: len = # of bytes
- PHISH_STRING: len = # of bytes, including the trailing NULL
- PHISH_INT*_ARRAY: len = # of int8 or int16 or int32 or int64 values
- PHISH_UINT*_ARRAY: len = # of uint8 or uint16 or uint32 or uint64 values
- PHISH_FLOAT_ARRAY: len = # of float values
- PHISH_DOUBLE_ARRAY: len = # of double values
- PHISH_PICKLE = len = # of bytes

The phish_datum() function returns information about the most recently received datum.

If *flag* is set to 0, phish_datum returns the number of fields in the datum. This value is also passed as an argument to the callback function invovked by the phish_loop and phish_probe functions, so a minnow typically does not need to use phish_datum to retrieve this info.

If *flag* is set to 1, phish_datum returns the input port the datum was received on. See the phish_port functions for a discussion of ports.

The phish_datum() function does not conflict with the phish_unpack() function. Phish_datum() can be called before or after or in between a series of phish_unpack() calls.

Restrictions: none

Related commands:

phish_recv, phish_pack

print minnow

Syntax:

```
print -f filename
```

- -f = optional switch for writing to a file
- filename = name of file to write to

Examples:

```
print
print -f outfile
```

Description:

Print is a PHISH minnow that can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The print minnow prints the datums it receives to stdout or to a file.

Ports:

The print minnow uses one input port 0 to receive datums. It does not use any output ports.

Operation:

When it starts, the print minnow opens *outfile* if it has been specified. It then calls the phish_loop function. Each time a datum is received on input port 0, its fields are looped over. Each field is written in the appropriate format with a trailing space, either to the screen or to *outfile*. A trailing newline is written after all the fields have been written.

The print minnow shuts down when its input port is closed by receiving a sufficient number of "done" messages. Before shutting down it closes *outfile* if it was specified.

Data:

The print minnow can receive datums with any number of fields. Any type of field can be printed, except for a field of type PHISH_RAW, which is ignored. Array-type fields are printed one value at a time, with trailing spaces.

Restrictions: none

Related minnows: none

reverse program

Syntax:

```
reverse
```

- this program takes no arguments

Examples:

```
wrapsink "reverse"
wrapss "reverse"
```

Description:

Reverse is a stand-along non-PHISH program that can be wrapped with a PHISH minnow so it can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The reverse program simply reads lines from stdin, reverses the order of the characters, and writes the resulting string to stdout. PHISH minnows that can wrap the reverse program include the wrapsink and wrapss, which convert stdin/stdout into the receiving and sending of datums.

Ports:

The reverse program does not call the PHISH library and thus does not use PHISH ports directly. But if it is wrapped with the wrapsink or wrapss minnows then they use one input port 0 to receive datums which are then read by the reverse program via stdin. If it is wrapped with the wrapss minnow then it uses one output port 0 to send datums that are written to stdout by the reverse program.

Operation:

The reverse program simply reads a line of input from stdin, stores it as a string, reverse the order of characters in the string, and writes it to stdout. See the doc pages for the wrapsink or wrapss minnows for how they convert datums they receive to lines of text that the reverse program can read from stdin, and how they convert lines of text that the reverse program writes to stdout to datums they send.

Data:

The reverse program does not call the PHISH library and thus does not deal directly with PHISH data types.

Restrictions:

The C++ version of the reverse program allocates a buffer of size MAXLINE = 1024 bytes for reading a line from stdin. This can be changed (by editing minnow/reverse.cpp) if longer lines are needed.

Related programs:

echo

slowdown minnow

Syntax:

```
slowdown delta
```

- delta = delay in seconds

Examples:

```
slowdown 0.1
slowdown 1.0
```

Description:

Slowdown is a PHISH minnow that can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The slowdown minnow sends datums as it receives them, but insures successive datums are sent no more often than every *delta* seconds. This can be useful for debugging PHISH nets that process data quickly.

Ports:

The shutdown minnow uses one input port 0 to receive datums and one output port 0 to send datums.

Operation:

When it starts, the shutdown minnow calls the phish_loop function. Each time a datum is received on input port 0, the phish_timer function is called and the elapsed time since the last datum was sent is calculated. If it is less than *delta* seconds, the minnow "sleeps" until *delta* seconds have passed. It then sends the datum to its output port 0 and records the time at which the send occurred.

The count minnow shuts down when its input port is closed by receiving a sufficient number of "done" messages.

Data:

The shutdown minnow can receive any kind of datums; it simply re-sends them as-is.

Restrictions: none

Related minnows: none

sort minnow

Syntax:

```
sort N
```

- N = keep top N sorted values

Examples:

```
sort 20
```

Description:

Sort is a PHISH minnow that can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The sort minnow receives counts of strings which it stores in a list. When it shuts down it sorts the list by count, and sends the top *N* counts and their associated strings.

Ports:

The sort minnow uses one input port 0 to receive datums and one output port 0 to send datums.

Operation:

When it starts, the sort minnow calls the phish_loop function. Each time a datum is received on input port 0, its first field is a count and its 2nd a string. The count/string pairs are stored in an internal table. This is done via an STL "vector" in the C++ version of sort, and via a "list" in the Python version of sort.

The sort minnow shuts down when its input port is closed by receiving a sufficient number of "done" messages. This triggers the sort minnow to sort the list of count/string pairs it has received. It then sends the top *N* results as datums to its ouput port 0. Each datum contains two fields. The first field is the count, the second is the string.

Data:

The sort minnow must receive two-field datums of type (PHSIH_INT32, PHISH_STRING). It also send two-field datums of type (PHISH_INT32, PHISH_STRING).

Restrictions: none

Related minnows:

count

wrapsink minnow

Syntax:

```
wrapsink "program"
```

- program = shell command for launching executable program

Examples:

```
wrapsink "myexe"
wrapsource "myexe -n 3 -o outfile <in.script"
wrapsink "ls *"
```

Description:

Wrapsink is a PHISH minnow that can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The wrapsink minnow is used to wrap a non-PHISH application so that datums can be sent to it from other PHISH minnows as lines it reads from stdin. It is a mechanism for using non-PHISH applications as minnows in a PHISH net.

Ports:

The wrapsink minnow uses one input port 0 and no output ports.

Operation:

When the wrapsink minnow starts, the *program* argument is treated as a string that is executed as a command by the shell. As in the examples above *program* can be an executable program name or a shell command. It can include flags or redirection operators. If the string contains spaces, it should be enclosed in quotes in the PHISH input script so that it is treated as a single argument when the script is read by the bait.py tool.

After the wrapsink minnow launches the *program* command, it calls the phish_loop function. Each time an input datum is received, its single string field is written to the running *program* with a trailing newline, so that the *program* reads it as a line of input from stdin. The *program* may write to the screen or a file as often as it chooses, but its output is not captured by the wrapsink minnow.

The wrapsource minnow shuts down when its input port is closed by receiving a sufficient number of "done" messages. When this occurs, it closes the stdin pipe the running *program* is reading from, which should cause it to exit.

Data:

The wrapsink minnow must receive single field datums of type PHISH_STRING.

Restrictions:

The C++ version of the wrapsink minnow allocates a buffer of size MAXLINE = 1024 bytes for converting the PHISH_STRING fields of received datums into lines of input read from stdin by the wrapped program. This can be changed (by editing minnow/wrapsink.cpp) if longer lines are needed.

Related minnows:

wrapsource, wrapss

wrapsource minnow

Syntax:

```
wrapsource -f "program"
```

- -f = optional flag for substituting input datums into "program"
- program = shell command for launching executable program

Examples:

```
wrapsource "myexe"
wrapsource "myexe -n 3 -o outfile <in.script"
wrapsource "ls *"
wrapsource -f "wc %s"
wrapsource -f "myexe -n 3 -o outfile  <%s"
```

Description:

Wrapsource is a PHISH minnow that can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The wrapsource minnow is used to wrap a non-PHISH application so that the lines it writes to stdout can be sent as datums to other PHISH minnows. It is a mechanism for using non-PHISH applications as minnows in a PHISH net.

Ports:

The wrapsource minnow uses one input port 0 if the -f flag is specified, otherwise it uses no input ports. It uses one output port 0 to send datums.

Operation:

The wrapsource minnow has two modes of operation, depending on whether the -f flag is specified. In either case, the *program* argument is treated as a string that is executed as a command by the shell.

As in the examples above *program* can be an executable program name or a shell command. It can include flags or redirection operators. If the string contains spaces, it should be enclosed in quotes in the PHISH input script so that it is treated as a single argument when the script is read by the bait.py tool.

If no -f flag is specified, the wrapsouce minnow launches a single instance of the *program* command and reads the output it writes to stdout a line at a time.

If the -f flag is specified, the wrapsouce minnow calls the phish_loop function. Each time an input datum is received, its single string field is inserted in the *program* string, as a replacement for a "%s" that it is presumed to contain. This can be used, for example, to substitute a filename into the *program* string. The wrapsource minnow then launches the modified *program* command and reads the output it generates. When the program exits, control returns to phish_loop, and a new datum can be received. Thus over time, the wrapsource minnow may launch many instances of *program*.

Each time a line of output is read from the running *program* the wrapsource minnow sends it as a string (without the trailing newline) to its output port 0.

If no -f flag is specified, the wrapsource minnow calls phish_exit after the launched program exits. If -f is specified, the wrapsource minnow shuts down when its input port is closed by receiving a sufficient number of "done" messages.

Data:

If the -f flag is specified, the count minnow must receive single field datums of type PHISH_STRING. It sends single-field datums of type PHISH_STRING.

Restrictions:

The C++ version of the wrapsouce minnow allocates a buffer of size MAXLINE = 1024 bytes for reading lines of output written to stdout by the wrapped program. This can be changed (by editing minnow/wrapsource.cpp) if longer lines are needed.

Related minnows:

wrapsink, wrapss

wrapss minnow

Syntax:

```
wrapss -f "program"
```

- program = shell command for launching executable program

Examples:

```
wrapsource "myexe"
wrapsource "myexe -n 3 -o outfile <in.script"
wrapsource "echo"
```

Description:

Wrapss is a PHISH minnow that can be used in a PHISH program. In PHISH lingo, a "minnow" is a stand-alone application which makes calls to the PHISH library to exchange data with other PHISH minnows.

The wrapss minnow is used to wrap a non-PHISH application so that datums can be sent to it from other PHISH minnows as lines it reads from stdin, and lines it writes to stdout can be sent as datums to other minnows. It is a mechanism for using non-PHISH applications as minnows in a PHISH net.

Ports:

The wrapss minnow uses one input port 0 to receive datums and one output port 0 to send datums.

Operation:

When the wrapss minnow starts, the *program* argument is treated as a string that is executed as a command by the shell. As in the examples above *program* can be an executable program name or a shell command. It can include flags or redirection operators. If the string contains spaces, it should be enclosed in quotes in the PHISH input script so that it is treated as a single argument when the script is read by the bait.py tool.

After the wrapss minnow launches the *program* command, it calls the phish_probe function. Each time an input datum is received, its single string field is written to the running *program* with a trailing newline, so that the *program* reads it as a line of input from stdin. When no input datum is available, "phish_probe" returns control to the wrapss minnow which checks if there is any output that the running *program* has written to stdout. If there is, the wrapss minnow sends it as a string (without the trailing newline) to its output port 0.

Note that there is no requirement that the running *program* produce a line of output for every line of input it reads. It may for example, read all of its input, compute for a while, then produce all of its output. Or it may produce output as bursts of lines, after reading multiple input lines.

The wrapss minnow shuts down when its input port is closed by receiving a sufficient number of "done" messages. When this occurs, it closes the stdin pipe the running *program* is reading from, which should cause it to exit. The wrapss minnow reads all the final output produced by the running program until it exits and converts it into datums that it sends to its output port 0. It then calls phish_exit.

Data:

The wrapss minnow must receive single field datums of type PHISH_STRING. It also sends single-field datums of type PHISH_STRING.

Restrictions:

The C++ version of the wrapss minnow allocates a buffer of size MAXLINE = 1024 bytes for both converting the PHISH_STRING fields of received datums into lines of input read from stdin by the wrapped program, and for reading lines of output written to stdout by the wrapped program. This can be changed (by editing minnow/wrapss.cpp) if longer lines are needed.

Related minnows:

wrapsink, wrapsource