PHISH Documentation

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

Table of Contents

Table of Contents

Table of Contents

PHISH Library Documentation

Version info:

The PHISH "version" is the date when it was released, such as 1 Sept 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 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, 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 we created 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.

Parallelism can be achieved by using multiple copies of processes, each working on a part of the stream. 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 desinged with streaming 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 set of

setup tools that convert an input script into a runnable program and allow it to be easily launched in parallel.

Our goal in developing PHISH was to make it easier to process data, particularly streaming data, in parallel, even 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.

1.2 PHISH lingo

The name PHISH was also chosen because it evokes the image of many 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 an individual process
- school = a set of duplicate minnows, working (swimming) in a coordinated fashion
- net(work) = a PHISH program, typically consisting of multiple minnow schools, connected together to perform a calculation, as in the diagram below
- bait.py = a script for hooking schools of minnows together into a net
- wrapper = a Python wrapper for the PHISH library, included in the PHISH distribution

---

1.3 PHISH pheatures

The model described above is not unique to PHISH. Many programs provide a framework for moving chunks of data between computational tasks interconnected by "pipes" in a data-flow 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 DataMiner tool (correct name?) from IBM uses a similar dataflow model, and is designed for processing streaming data at high rates.

NOTE: need to doc,cite the IBM tool and Titan better

These programs include a suite of processing modules and are typically 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. This means data can be sent from one computational task to another in a low-overhead fashion by passing a pointer or via shared memory buffers. (Is the preceding true, does Titan have nodes that can do MPI-style paralellism?).

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

NOTE: could also contrast nodes of a pipeline working one at a time as data moves as a chunk down the pipeline (could exploit parallellism at each node, or one process could run the entire pipeline, one stage at a time), versus the PHISH minnows working continuously as data streams thru the pipeline continusously

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 (couple 1000 lines), with a compact API (couple dozen functions). It is written in C++, with a C-style interface, so that it can be easily called from programs written in a variety of languages (C, C++, Fortran, Python, etc). The library is highly portable and 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, one based on sockets. The latter uses the open-source ZMQ library. This means you need one or both of these packages (MPI, ZMQ) installed on your machine to build a program (minnow) that uses the PHISH library.
- A Python wrapper for the PHISH library is provided, so that programs (minnows) that call the PHISH library can be written in Python.
- The PHISH library encodes data exchanged between processes (minnows) with strict data typing rules, so that data can be passed between programs written in different languages (e.g. C++ vs Fortran vs Python) and running on different machines (4-byte vs 8-byte integers). Eventually, we may also allow for data exchange between machines with different floating point representations or endian ordering of data types.
- PHISH programs (nets) which involve coordinated computation and data exchange between many processes (minnows) can be specified in PHISH input scripts, which are text files with a simple command syntax.
- PHISH input scripts use a connect command which allows data to be exchanged in various patterns between collections of independent processes (minnows). This enables parallelism in data processing to be easily expressed and exploited.
- PHISH input scripts can be converted into launch scripts via a provided Bait.py program. This produces a file suitable for running either with MPI or sockets.
- PHISH programs (nets) running on top of MPI are launched via the standard mpirun or mpiexec command. Note that a PHISH net is different than the usual MPI program run on P processors where P copies of the same executable are launched. A PHISH net typically consists of several different schools of minnows; each minnow is an independent executable.
- PHISH programs (nets) running on top of sockets are launched via a provided Python script, called Launch.py, which mimics the operation of mpirun. It invokes the set of independent processes (minnows) on various machines via ssh commands, sets up the socket connections between them, and synchrnoizes their launch so that no data is dropped as the processes (minnows) begin exchanging data.
- PHISH programs (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 program (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 programs (nets) can be launched independently and exchange data. This is a mechanism for adding/deleting processes (minnows) to/from a calculation on the fly.
- A handful of programs (minnows) that call the PHISH library are included in the distribution, as are some example PHISH input scripts that encode PHISH programs (nets). Makefiles are also provided to assist in creating and building your own new programs (minnows).
- Programs (minnows) are included that wrap non-PHISH applications that read from stdin and/or write to stdout. This allows those executables to be used in a PHISH program (net) and exchange data with other programs (minnows).

1.4 Steps to creating and running a PHISH net

The PHISH package contains a library and several related tools for defining and running PHISH nets. These are the steps and associated tools 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 chunks of 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 defining a PHISH net, as schools of minnows and the communication patterns connecting them
4. uss the bait.py tool to process the input script
5. use mpirun or Launch.py to run the output file created by Bait.py,
6. and perform the calculation

Step (1): An overview of the PHISH library and instructions for how to build it is given in this section.

Step (2): A minnow is a stand-alone application 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 to 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 commands used in PHISH input scripts are described in this section, with links to individual commands that explain their operation and options.

Step (4): The bait.py tool, its command-line options, and instructions on how to run it, are described in this section.

Step (5): Bait.py produces a file as output. If the "-mode mpich" or "-mode openmpi" switch was used, the file can be run using the standard mpirun command to run a parallel job. In all the examples that follow, "outfile" is the name of the file produced by bait.py.

For MPICH, you would do this as follows:

```
mpiexec -configfile outfile
```

For OpenMPI, you would do this as follows:

```
mpirun -configfile outfile
```

If the "-mode socket" switch was used with bait.py, then the file can be run using the Launch.py tool, as follows:

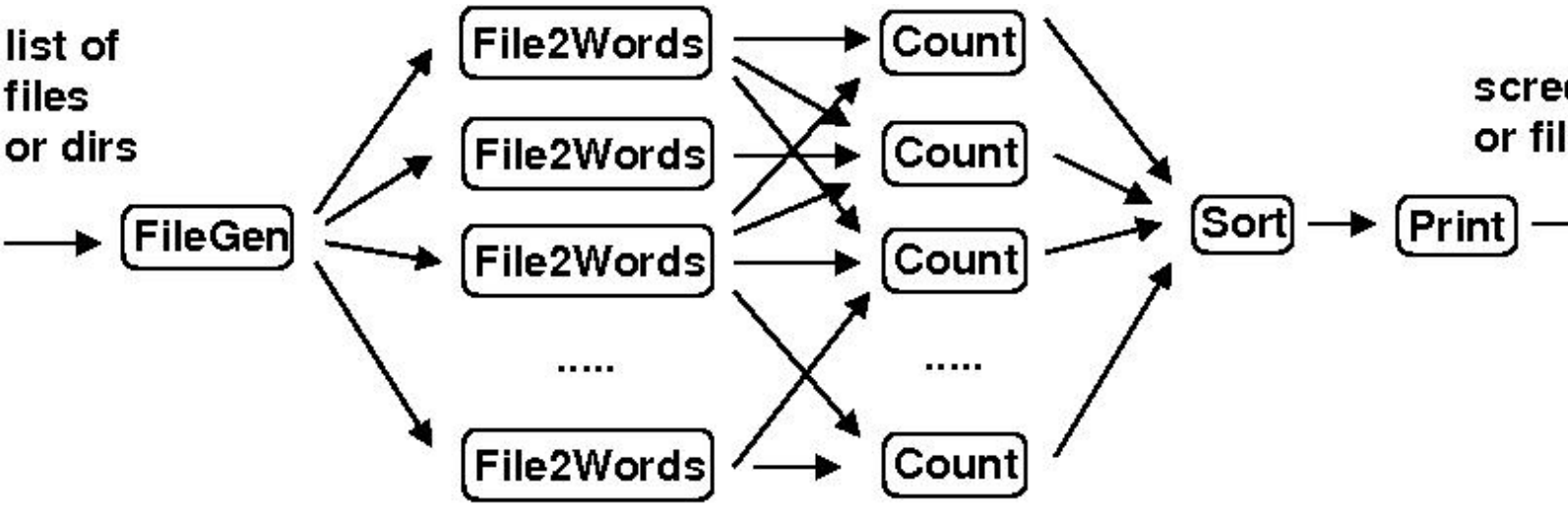
```
launch.py ... outfile
```

TIM: Launch.py support the following command-line arguments ...

TIM: Does launch.py need its own doc page?

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 beconnected 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. The key point is that each *Count* minnow will receive all occurrences of a subset of possible words. It stores an internal hash table and counts the occurrences of each word it receives.

When the *FileGen* minnow sends the last filename if finds, it sends a "done" message to each of the *File2Words* minnows. When they receive a "done" message, they send a "done" message to each *Count* minnow. When a *Count* minnow receives 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 a list of 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, one by one, 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

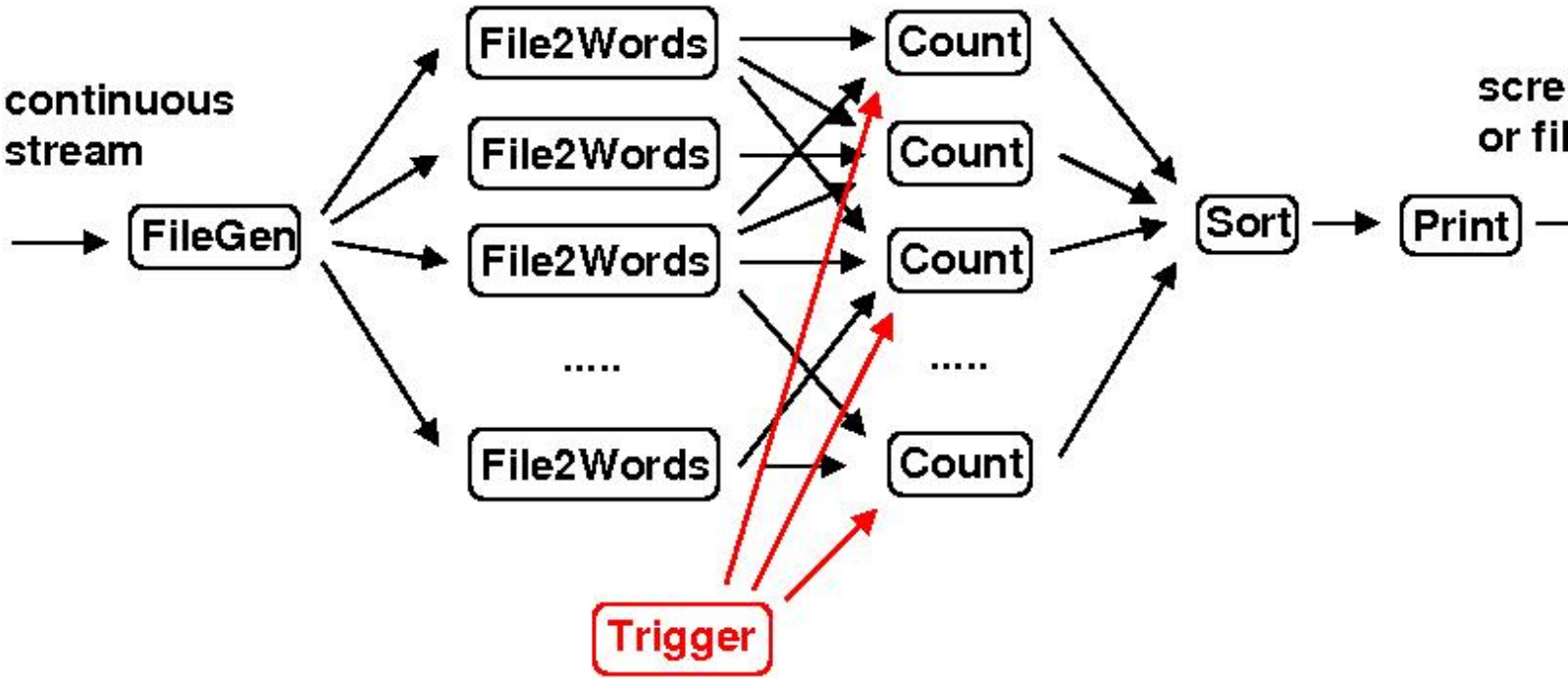
each "reduce" process (performed by the *Count* minnows). This is the "shuffle" phase of a Hadoop MapReduce. The reducer performs a merge sort of all the words in the chunks it receives (one from each mapper). It can then calculate the count for each unique word.

In contrast, the PHISH program operates in a more continuous 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. In this example, 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.

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 describe 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, PHISH in action, J Parallel and Distributed Compuing, submitted (2012).

PHISH was developed by the following individuals at Sandia:

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

2. Bait.py Tool

Bait.py is a Python program which converts a PHISH input script into another file that can be launched via MPI or the launch.py tool, to run a PHISH net and perform a calculation. 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 minnows.

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 create a new launch 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

- variable
- set
- minnow
- connect
- layout

2.2 Running bait.py

The bait.py Python script is 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.

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.

-var or -v	-var name str1 str2 ...
-out or -o	-out filename
-path or -p	-path dir1:dir2:dir3:...
-mode or -m	-mode outstyle

The *-var* switch defines a variable that can be used within the script. It can be used multiple times to define different variables. A variable command can also be used in the input script itself. The variable name is any alphanumeric string. A list of strings is assigned to it, e.g. a series of filenames. For example,

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

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

The *-out* switch specifies a filename that bait.py will create when it writes out the MPI or socket script that can be used to launch the PHISH program. The default value is "outfile".

The *-path* switch specifies a colon-separated list of directories, which are added to an internal list stored by bait.py. Initially the list contains only the current working directory. When bait.py processes each minnow, as specified by the minnow command, it looks for the minnow's executable file in the list of directories, so that it can write it to the launch script with a full, correct path name. This switch can be used multiple times, adding more directories each time.

The *-mode* switch specifies the format of the launch file that bait.py writes out. The valid *outstyle* values are *mpich* or *openmpi* or *socket*. *Mpich* is the default.

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:

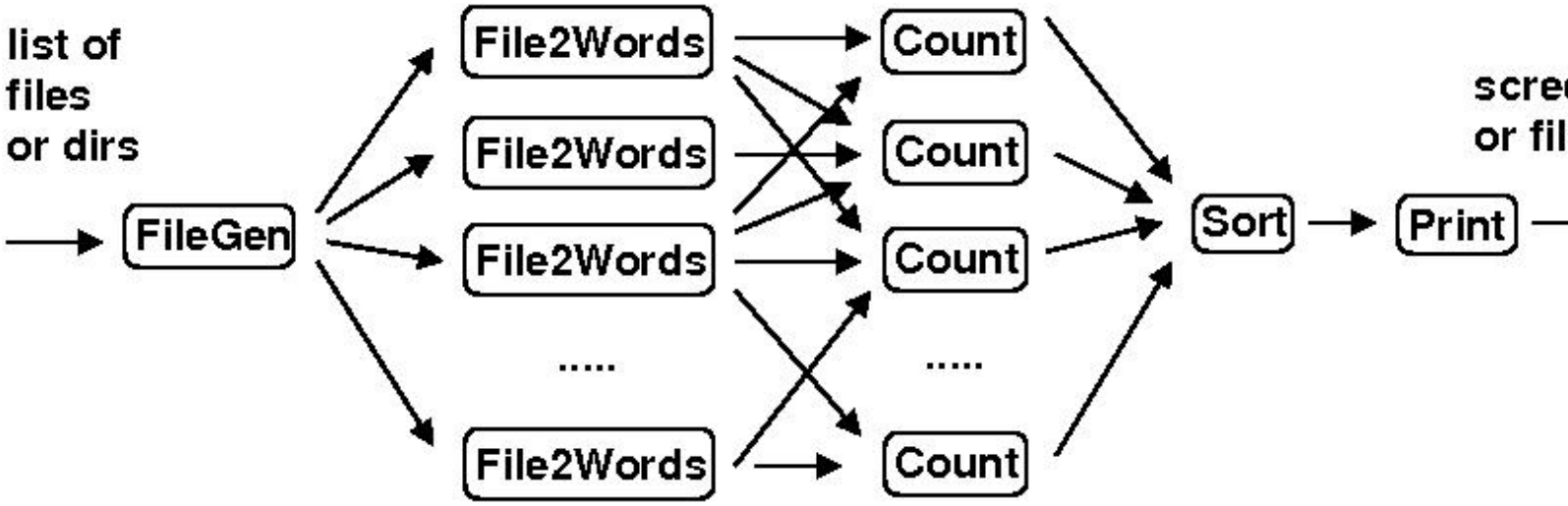
```
minnow 1 filegen f1.txt f2.txt f3.txt
```

After variable substitution, a single command is a series of "words" separated by whitespace. The first word is the command name; the reamining 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 converted by bait.py into a launch script for running a PHISH program, after the entire script is read. The exception is that a variable cannot be substituted for 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 example/in.wc 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

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

layout 1 1
layout 2 5
layout 3 3
layout 4 1
```

```
layout 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 connect commands specify the communication pattern used bewteen different sets 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 layout 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 -v files in.* -p ../minnow <in.wc
```

then bait.py produces the following lines in outfile

-n 1 ../minnow/filegen -minnow filegen 1 1 0 -out 1 0 0 roundrobin 5 1 0 -args in.bottle in.cc in.cc.jon in.filelist in.pp in.rmat in.slow in.wc in.wrapsink in.wrapsource in.wrapsourcefile in.wrapss -n 5 ../minnow/file2words -minnow file2words 2 5 1 -in 1 0 0 roundrobin 5 1 0 -out 5 1 0 hashed 3 6 0 -n 3 ../minnow/count -minnow count 3 3 6 -in 5 1 0 hashed 3 6 0 -out 3 6 0 single 1 9 0 -n 1 ../minnow/sort -minnow sort 4 1 9 -in 3 6 0 single 1 9 0 -out 1 9 0 single 1 10 0 -args 10

```
-n 1 ../minnow/print -minnow print 5 1 10 -in 1 9 0 single 1 10 0
```

which is the format of a "configfile" for the MPICH flavor of MPI. 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. Several switches like "-minnow", "-in", "-out" follow which are created by bait.py to encode the communication patterns between the minnows as represented by the diagram above and the connect commands of the input script. The final "-args" switch is followed by minnow-specfic arguments that appeared in the input script.

As discussed in this section of the Introduction doc page, this outfile can be launched via the MPICH mpiexec command as:

```
mpiexec -configfile outfile
```

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 version 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:

NOTE: should we include file2fields in the distro?

- count
- file2words
- filegen
- ping
- pong
- print
- rmat
- slowdown
- sort

These are 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 simple codes which can be compiled into stand-alone non-PHISH executables. They are 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 cases 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
2
3   import sys,os,glob,copy
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("File2words processes string values")
11    if hash.has_key(str): hashstr = hashstr + 1
12    else: hashstr = 1
13
14  def sort():
15    pairs = hash.items()
16    for key,value in pairs:
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,sort,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 build this module, which wraps the C-interface to the PHISH library, and add it to your Python 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, as described in this section, extra PHISH library command-line arguments are used 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 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 connection 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 functino is called. This turns control over 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 sort() 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 sort() 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 sort() 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 communication channels by which a minnow receives datums from an upstream minnow or sends datums to a downstream minnow.

Any minnow can define and use 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 PHISH 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 connections that are specified and the number of minnows on the other end of each connection. Thus it is possible for the user to specify connections 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 possible and 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. These is because these are really settings made 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 should 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:

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 connect 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 attempts to not lose messages in this scenario, with the effect that the overall pipeline of processing naturally throttles itself to the rate of the bottlenecking minnow. This is handled by the underlying MPI or socket message passing protocols. 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 the "safe" mode of data exchange which can be enabled by the set safe command in a PHISH input script. This will use MPI_Ssend() calls which enforce extra handshaking between the sending and receiving processes to avoid dropping messages.

Note that if the PHISH net is processing real=time data in a continuous streaming mode, and one or more PHISH minnows cannot keep up with the incoming data rate, then the minnow that ingests the real time stream has no choice but to drop data at some point.

TIM: is there a socket analog to this issue?

3.4 Shutting down a minnow

PHISH minnows can be designed to process a finite or infinite stream of data. In the latter case, the PHISH net of minnows is typically shut down by the user killing one or more of the processes. In the former case, you often want each minnow in the net to shut down cleanly. The PHISH library sends special "done" messages when the 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_inpu .htmlt 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. An exception is

when a school of minnows exchanges data in a "ring" style of commuication as setup by the connect ring command in a PHISH input script. In this case, 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.

NOTE: doc what happens if Ctrl-C out of mpirun

NOTE: doc that shutdown in ZMQ is buggy

3.5 Building a minnow

Minnows are stand-alone programs which simply need to be linked with the PHISH library. New 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.

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 work for 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.a or libphish-zmq.a.

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.

If a new minnow is more complex (e.g. multiple files), you could 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 this final build step.

Your executable minnow files do not need to be put in 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 is 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 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 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. The doc pages for individual library functions document all 3 APIs; they are very similar. 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. They 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()
   phish_send_direct()

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.

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.

If none of the provided files work for 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.

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*.a 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");
```

NOTE: should namespace be PHISH or phish or Phish? change in phish.hpp accordingly

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, the Python PHISH wrapper needs to be installed in your machine's Python. See this section of the manual for details. A Python program can then invoke a library function as

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

or

```
from import phish *
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 of data. 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 time within the PHISH library.

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

- # 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 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_RAW = 0
- PHISH_CHAR = 1
- PHISH_INT8 = 2
- PHISH_INT16 = 3
- PHISH_INT32 = 4
- PHISH_INT64 = 5
- PHISH_UINT8 = 6
- PHISH_UINT16 = 7
- PHISH_UINT32 = 8
- PHISH_UINT64 = 9
- PHISH_FLOAT = 10
- PHISH_DOUBLE = 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_RAW is a string of raw bytes, which can be of any length, and which the minnow can format in any manner. PHISH_CHAR, PHISH_INT*, PHISH_UINT*, PHISH_FLOAT, and PHISH_DOUBLE are a single character, signed integer (of length 8,16,32,64 bits), unsigned integer (of length 8,16,32,64 bits), a float (typically 4 bytes), and double (typically 8 bytes). PHISH_STRING is a standard C-style NULL-terminated C-string. The NULL is included in the field. The ARRAYS are contiguous sequences of int*, uint*, float, or double values. PHISH_PICKLE is used by 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.

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 (defined at the top of src/phish.cpp) is added to the tag.

NOTE: MAXPORT is hardwired, could re-compile - this is so all minnows use same setting.

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.

TIM: How is this encoding of port and done implemented for sockets?

5. Examples

Below is the list of PHISH input scripts provided in the examples directory of the distribution. Many of them come in two flavors, using minnows written in C++, or minnows written in Python. The scripts can also be edited to use a mixture of C++ and Python minnows in the same PHISH net. See the comment lines at the top of each script for instructions on what variables they expect to be defined via command-line arguements to the bait.py tool.

As discussed in this section of the Introduction, the bait.py tool produces an output file. The output file can be launched as a parallel job by the MPI commands mpirun or mpiexec, if the minnows specified in the input script were built with the MPI version of the PHISH library. Or the output file can be used with the "launch.py" tool to run a parallel job if the minnows were built with the socket version of the PHISH library.

---

in.pp

in.pp.py

These scripts ping-pong a message back-and-forth between 2 minnows, ping and pong. Each minnow is connected to the other and they exchange a M byte message N times.

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

```
../bait/bait.py -v m 1000 -v n 100 -p ../minnow <in.pp
```

can M and N be defined with default values in input script?

---

in.wc

in.wc.py

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

The scripts can perfrom the operation in parallel by changing the size of the school of readfile and count minnows in the layout commands within the input script. They can process a large corpus of files by specifying one or more directory names as arguments to the filegen minnow.

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

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

make Nmap and Nreduce variables

---

in.wrapsink

in.wrapsink.py

These scripts demonstrate the use of the wrapsink minnow which is used to wrap a non-PHISH application. The program reads from stdin to receive messages sent to it from other minnows, and writes directly to stdout.

In this case the "reverse" program in the minnows directory is wrapped and it reverses each filename in a list of input filenames. Multiple copies of reverse can be launched by changing the size of the school of wrapsink minnows in the layout commands within the input script.

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

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

make layout arg a variable

---

in.wrapss

in.wrapss.py

These scripts are similar to the preceding in.wrapsink scripts. They reverse each filename in a list of input filenames. In this case the reverse program is wrapped with the wrapss tool which lets the program read from stdin to receive messages sent to it from other minnows, and sends the output it writes to stdout as messages to downstream minnows, in this case the print minnow. As before, multiple copies of reverse can be launched by changing the size of the school of wrapss minnows in the layout commands within the input script.

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

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

make layout arg(s) a variable

---

in.wrapsource

in.wrapsource.py

what does this do? does it exist?

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.

Before using the PHISH library in a Python script, the Python on your machine must be "extended" to include an interface to the PHISH library. 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

The PHISH library has two variants (both have the same API), one that uses the MPI message-passing library, the other that uses the socket libarary ZMQ. To use either variant from Python you must have either MPI or ZMQ installed on your machine.

From the python directory of the distribution, type one of these commands:

```
python setup_mpich.py build
python setup_openmpi.py build
python setup_zmq.py build
```

and then the corresponding version of one of these commands:

```
sudo python setup_mpich.py install
python setup_mpich.py install --home=~/foo
```

The "build" command should compile all the needed PHISH C++ files. The first "install" command will put the needed files in your Python's site-packages sub-directory, so that Python can load them. For example, if you installed Python yourself on a Linux machine, it would typically be somewhere like /usr/local/lib/python2.7/site-packages. Installing Python packages this way often requires you to be able to write to the Python directories, which may require root priveleges, hence the "sudo" prefix. If this is not the case, you can drop the "sudo".

Alternatively, you can install the PHISH files (or any other Python packages) in your own user space. The second "install" command does this, where you should replace "foo" with your directory of choice.

If these commands are successful, a *phish.py* and *_phish.so* file will be put in the appropriate directory.

NOTE: say something about needing Python 2.7 for valid ctypes?

Creating a shared MPI library

A shared library is one that is dynamically loadable, which is what Python requires. On Linux this is a library file that ends in ".so", not ".a". Such a shared library is normally not built if you installed MPI 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 use "sudo make install" in place of the last line. The end result should be the file libmpich.so in /usr/local/lib. Note that if the file libmpich.a already existed in /usr/local/lib, you will now have both a static and shared MPICH library. This will be fine for Python PHISH since it only uses the shared library. But if you build other codes with libmpich.a, then those builds may fail if the linker uses libmpich.so instead, unless other dynamic libraries are also linked to.

Testing the PHISH library from Python

Before importing the PHISH library in a Python program, one more step is needed. The interface to the library is via Python ctypes, which loads the shared PHISH library via a CDLL() call, which in turn is a wrapper on the C-library dlopen(). This command is different than a normal Python "import" and needs to be able to find the PHISH shared library, which is either in the Python site-packages directory or in a local directory you specified in the "python setup.py install" command, as described above.

The simplest way to do this is add a line like this to your .cshrc or other shell start-up file.

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/usr/local/lib/python2.7/site-packages
```

and then execute the file to insure the path has been updated. This will extend the path that dlopen() uses to look for shared libraries.

To test if the PHISH library has been successfully installed, launch python in serial and type

```
>>> import phish
```

If you get no errors, you're good to go.

7. Errors

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

Debugguing 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 the bug if something goes wrong:

As with any parallel program, running it 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 layout command in the PHISH input script.
In principle, a PHISH net should behave similarly whether it is run entirely on one desktop machine, on a large parallel machine, or on a distribud network. The first case, running all minnows on a single desktop machine, is usually the easiest mode to debug in.
One or more "slowdown" minnows can be inserted in a pipeline 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 connect 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 use of a minnow's output port by an input script is optional, as discussed in this section, so that downstream minnows can often be commented out to focus debugging on earlier stages of the data processing pipeline.
Alternatively, 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 encoutered a message in the following format is printed to stderr:

```
PHISH MPI ERROR: Minnow str1 ID str2 # N: message
```

if the MPI version of the PHISH library is being used; MPI is replaced by SOCKET if the socket version is being used.

NOTE: say something about whether error causes an exit or returns to the calling minnow

Str1 is the name of the minnow executable (e.g. count). Str2 is the ID of the minnow as specified in the PHISH input script via the "minnow" command. N is the global ID of the minnow process. This is a number from 0 to Nglobal-1 where Nglobal = the total number of minnows in the PHISH net as specified by the input script. This

information may be useful in debugging the minnow or PHISH input script.

If a warning condition is detected, the same format of message is printed, with ERROR replaced by WARNING.

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-mpi.cpp or src/phish-socket.cpp can be searched for the message text.

Error messages from bait.py

As discussed in this section, the bait.py tool is used to convert PHISH input scripts into an output file that is used to launch a PHISH run. Any errors it encounters in the input script generate an error message of this form:

```
Bait.py error: message
```

NOTE: check this syntax.

The messages 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.

Error messages from bait.py

As discussed in this section, the launch.py tool is used to run a PHISH net whose minnows are built with the socket version of the PHISH library. Any errors it encounters at launch time generate an error message of this form:

```
Launch.py error: message
```

NOTE: check this syntax.

The messages should be self-explanatory. See the doc page for the launch.py tool for relevant details. If necessary, the Python source code in python/launch.py can be searched for the message text.

TIM: check this section

layout command

Syntax:

connect 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 = kind of connection between sending minnows and receiving
- 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:

```
connect ...
```

Description:

Connect 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 one minnow is routed to the input of another minnow 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 connections defined by a series of connect commands defines how a school of minnows is harnessed together to perform a desired computational task. It also defins how parallelism is exploited by the collection of minnows.

A connection is made between two sets of minnows, one set sends datums, the other set receives them. Each set may contain one or more processes, where a process is a single minnow. The layout command specifies how many minnows run within each set. Since a datum is typically sent from a single minnow to a single receiving minnow, the style of the connection determines which minnow in the sending set communicates with which minnow in the receiving set.

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 connections, and for a PHISH input script to connect 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 connect commands which connect the same *sendID* and same (or different) *outport* to different *recvID:inport* minnows. Likewise, there can be multiple connect commands which connect the same *recvID* and same (or different) *inport* to different *sendID:outport* minnows. There can even be multiple connect commands which connect 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 connection styles supported by the connect command.

The *single* style connects *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 connects *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 connects *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 connects *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 connects *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 sent to. See the doc page for the phish_send_direct() library function for more explanation of how this is done.

The *bcast* style connects *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 *publish* and *subscribe* styles are different in that they do not connect two sets of minnows to each other. Instead they connect 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 connects *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 connects *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 multple 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.

NOTE: how does the PHISH library check the socket and return if there is no datum?

NOTE: how does the read from the socket delimit the PHISH datum, so the minnow knows how much to read?

Restrictions:

The *publish* and *subscribe* styles are only supported by the socket version of the PHISH library, not the MPI version.

Related commands:

minnow, layout

Default: none

layout command

Syntax:

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

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

```
possible keywords = host or invoke
  host value = machine
    machine = name of host to run the minnow on
              what if want some minnows on one host, some on another?
  invoke value = launcher
    launcher = run minnow via this program
```

Examples:

```
layout 3 10
layout countapp 1
layout countapp 1 host foo.locallan.gov
layout myApp 5 invoke python
```

Description:

Layout 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 *host* keyword sets a value used by the socket version of the PHISH library, and thus can only be used when the *-mode socket* command-line argument is used with bait.py. The *machine* value is the name of the machine (e.g. foo.locallan.gov) to launch all the minnow processes on.

The *invoke* keyword can be used if the minnow application should be run by another program. For example if the minnow is a Python script, the *launcher* could be set to "python" or to "/usr/local/bin/python2.4". In this case, if the minnow *exefile* was specified as foo.py, then the launch script output by bait.py would include a line such as

```
python foo.py ...
```

instead of simply

```
foo.py ...
```

Restrictions: none

Related commands:

minnow

Default:

If a layout command is not specified, then Np is assumed to be 1, so that one process is launched when the PHISH program is run.

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:

layout

set command

Syntax:

set keyword value

- keyword = *memory* or *safe* or *port*

```
memory value = N
    N = max size of datum in Kbytes
  safe value = none
```

Examples:

set memory 1024 set safe

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. Only 2 such buffers are allocated, so this setting essentially determines the memory footprint of 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 *safe* keyword is only relevant the MPI version of the PHISH library. It forces the library to use MPI_Ssend() calls which are a safer version than the normal MPI_Send() function. 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. With the safe mode of MPI calls, the slow minnow should effectively throttle the incoming messages so an overflow does not occur. This requires extra handshaking between the MPI processes and slows down the rate at which small messages are exchanged, so this safe mode is "off" by default. Many PHISH programs do not seem to need it, as MPI is robust enough to insure no messages are dropped.

Note that the *safe* keyword takes no value. If it is not specified, the default is for the PHISH library to use normal MPI_Send() calls.

Restrictions: none

Related commands:

See the discussion of command-line arguments for the bait.py tool.

Default:

The default settings are memory = 1 (1 Kbyte), safe is not set.

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 is 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.

Restrictions: none

Related commands:

See the *-var* command-line argument for bait.py.

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 is 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 send 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: none

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

file2fields minnow

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 ping 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)())
```

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)())
```

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 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_error 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 flag) { }
```

or

```
def abortfunc(flag)
```

in Python,

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

As explained on the phish_error doc page, the phish_error() function prints a message and then 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.

TIM: what is the flag?

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: none

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 error(char *str)
void warn(char *str)
void 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:

There 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 minnows input or output ports. The latter 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, invokes the user-specified abort callback function if it is defined via phish_callback, then calls phish_abort().

The error message is printed in this format for the MPI version of the MPI library (replated MPI by SOCKET for the socket version):

```
PHISH MPI ERROR: Minnow exename ID idminnow # idglobal: message
```

where exename is the name of executable minnow file (not the full path, just the filename), idminnow is the ID of the minnow as specified in the PHISH input script, idglobal 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 school of minnows 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. The abort callback is not invoked and neither is phish_abort(). Control is simply returned to the calling minnow which can continue on.

The phish_abort() function prints no message.

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

TIM: how does this work for socket version? Does it cause all minnows to 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 connect 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 connect 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:

```
void phish_init(int *narg, char ***args)
```

C examples:

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

C++ syntax:

```
void init(int *narg, char ***args)
```

C++ examples:

```
PHISH::init(&argc,&argv);
```

Python syntax:

```
def init(args)
```

Python examples:

```
import phish
args = phish.init(sys.argv)
first_minnow_arg = args[0]
```

Description:

This is a 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.

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.

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 MPI or socket launch script that the bait.py tool creates from 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 reamining minnow-specific arguments, which the minnow can read and process.

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.

There are the switches and arguments the PHISH library looks for and processes. These are generated automatically by the bait.py tool when it processes a PHISH input script, so normally you don't need to think about this level of detail, but it may be helpful for understanding how PHISH works.

- -minnow exefile ID Nlocal Nprev
- -memory N
- -safe
- -in sprocs sfirst sport style rprocs rfirst rport
- -out sprocs sfirst sport style rprocs rfirst rport
- -args arg1 arg2 ... = args for the minnoq itself

The *-minnow* switch appears once, as the first argument. *Exefile* is the name of executable file for this minnow, e.g. count or count.py. The ID is the minnow 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 *-memory* and *-safe* and switches change default settings within the PHISH library.

The *-memory* value *N* sets the maximum size of the buffers used to send and receive datums.

The *-safe* switch forces the MPI version of the PHISH library to use MPI_SSend() calls instead of the standard MPI_Send(). These are "safer" in the sense they insure messages are not dropped due to a minnow not keeping up with its incoming messages. See the set command of the bait.py tool for more information on the settings of these switches.

The *-in* switch appears once for every connection the minnow has with other minnows, where it is a receiver of datums. See the connect 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 connect 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 *-out* switch appears once for every connection the minnow has with other minnows, where it is a sender of datums. See the connect 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 connect 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 *-args* switch appears last and lists all the remaining minnow-specific arguments. The PHISH library ignores these, but strips of all command-line arguments up to and including the *-args* switch before returning the args to the minnow caller.

The phish_init() function also flags each specified input port and output port with a CLOSED status, instead of UNUSED. See the connect 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.

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_raw(char *buf, int32_t n);
void pack_char(char value);
void pack_int8(int8_t value);
void pack_int16(int16_t value);
void pack_int32(int32_t value);
void pack_int64(int64_t value);
void pack_uint8(uint8_t value);
void pack_uint16(uint16_t value);
```

```
void pack_uint32(uint32_t value);
void pack_uint64(uint64_t value);
void pack_float(float value);
void pack_double(double value);
void pack_string(char *str);
void pack_int8_array(int8_t *vec, int32_t n);
void pack_int16_array(int16_t *vec, int32_t n);
void pack_int32_array(int32_t *vec, int32_t n);
void pack_int64_array(int64_t *vec, int32_t n);
void pack_int8_array(int8_t *vec, int32_t n);
void pack_int16_array(int16_t *vec, int32_t n);
void pack_int32_array(int32_t *vec, int32_t n);
void pack_int64_array(int64_t *vec, int32_t n);
void pack_float_array(float *vec, int32_t n);
void pack_double_array(double *vec, int32_t n);
void pack_pickle(char *buf, int32_t n);
```

C++ examples:

```
#include "phish.hpp"
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);
```

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 exepct 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 iport, void (*datumfunc)(int), void (*donefunc)(), reqflag)
void output(int iport)
```

C++ examples:

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

There 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 connect 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 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 connect 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.

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_done 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 "datumfunc" 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 connect 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 connect 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 connect command for details. If one or more of the connection styles connected to the 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.

NOTE: does Python syntax not include len? or can other non-string keys be *hashed*?

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 connect 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 performed by invoking MPI_Finalize(), which means no further MPI calls can be made.

NOTE: how is this done for the sockets version?

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 less often used 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. For the MPI version of the PHISH library, the function is a wrapper on MPI_Wtime().

NOTE: how does this work for socket version?

The function returns the current time in CPU seconds. To calculated an elapsed time, you need to bracket a section of code with 2 calls to phish_timer() and calcualte 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_RAW = 0
- PHISH_CHAR = 1
- PHISH_INT8 = 2
- PHISH_INT16 = 3
- PHISH_INT32 = 4
- PHISH_INT64 = 5
- PHISH_UINT8 = 6
- PHISH_UINT16 = 7
- PHISH_UINT32 = 8
- PHISH_UINT64 = 9
- PHISH_FLOAT = 10
- PHISH_DOUBLE = 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_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_INT32 is a signed 32-bit integer. PHISH_UINT64 is an unsigned 64-bit int. PHISH_DOUBLE is a double-precision floating point value, typically 64-bits in length. PHISH_STRING is a standard C-style NULL-terminated string. The ARRAY types mean the field is a sequence of "int" or "uint" or "float" or "double" values, packed one after the other.

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_RAW: len = # of bytes
- PHISH_CHAR: len = 1
- PHISH_INT*: len = 1
- PHISH_UINT*: len = 1
- PHISH_FLOAT: len = 1
- PHISH_DOUBLE: len = 1
- 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

Note that the PHISH_PICKLE flag is only used by the Python interface to the PHISH library to encode Python data types. It should not normally be used in a minnow written in C or C++.

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

ping minnow

Syntax:

```
ping N M
```

- N = # of datums to ping/pong with partner minnow
- M = # of bytes in each datum

Examples:

```
ping 1000 0
ping 100 100000
```

Description:

Ping 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 ping minnow is designed for use with the pong minnow, to exchange datums back and forth.

Ports:

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

Operation:

When it starts, the ping minnow creates a buffer *M* bytes long and fills it with NULLs (zeroes). It sends this buffer to its output port 0. It then calls the phish_loop function. Each time a datum is received on input port 0 (i.e. from the pong minnow), it is re-sent to output port 0.

When the ping minnow receives a datum for the *N*th time, it calls the phish_exit function to shut down. It also prints the elapsed time for exchanging an *M*-byte datum *N* times.

Data:

The first datum the ping minnow sends has a single field of type PHISH_RAW. It does not care what kinds of datums it receives and re-sends.

Restrictions:

The default buffer size for sending and receiving datums is 1 Kbyte. To use an *M* value that exceeds this, use the "set memory" command should be used in the PHISH input script so that the PHISH library allocates larger buffers.

Related minnows:

pong

pong minnow

Syntax:

```
pong
```

- this minnow takes no arguments

Examples:

```
pong
```

Description:

Pong 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 pong minnow is designed for use with the ping minnow, to exchange datums back and forth.

Ports:

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

Operation:

When it starts, the pong minnow calls the phish_loop function. Each time a datum is received on input port 0 (i.e. from the ping minnow), it is re-sent to output port 0.

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

Data:

The pong minnow does not care what kinds of datums it receives and re-sends.

Restrictions:

The default buffer size for sending and receiving datums is 1 Kbyte. To use an *M* value that exceeds this, use the "set memory" command should be used in the PHISH input script so that the PHISH library allocates larger buffers.

Related minnows:

ping

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

NOTE: print minnow needs some additional logic for various added field types **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: none

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/echo.cpp) if longer lines are needed.

Related programs:

echo

rmat minnow

Syntax:

```
rmat N Nlevel a b c d fraction seed
```

- N = # of matrix elements or edges to generate
- Nlevel = order of RMAT matrix = 2^Nlevel
- a,b,c,d = quadrant weighting factors which sum to 1.0
- fraction = twiddle factor
- seed = random number generator seed (positive integer)

Examples:

```
rmat 10000 20 0.25 0.25 0.25 0.25 0.1 12345
rmat 1000000 30 0.45 0.25 0.25 0.05 0.0 8787843
```

Description:

Rmat 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 rmat minnow generates N elements from an RMAT matrix of order 2^*Nlevel* in a random recursive fashion. An RMAT matrix is a randomized sparse matrix whose structure can be tailored by the input parameters *a*, *b*, *c*, and *d*. The matrix is of order 2^*Nlevel* and an element is represented by the I,J indices of the row and column it is in. The I,J values can also be interpreted as an edge in a sparse graph between vertices I and J, where the vertices are numbered from 1 to 2^*Nlevel* inclusive.

The utility of RMAT matrices for describing graphs of various kinds is discussed in (Smith).

Ports:

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

Operation:

The rmat minnow generates *N* matrix elements and sends each as a datum to its output port 0. Each datum has 2 fields, which are the I and J of the row and column of the matrix element. I and J are both integers from 1 to 2^*Nlevel* inclusive.

The method of generating I and J is recursive with *Nlevel* levels. At each stage one of 4 quadrants of the "current" matrix is selected randomly with relative probablities *a*, *b*, *c*, and *d*. At the first level, the current matrix is the entire matrix. After selecting a quadrant, that quadrant becomes the current matrix. Thus as the last level, a single I,J element is selected out of the original full matrix. The *twiddle* factor is used at each recursion level to adjust *a*, *b*, *c*, and *d* by a random fractional amount while keeping the sum of the 4 values = 1.0. This has the effect of further randomizing the distribution of generated matrix elements.

NOTE: better def of twiddle and is it 0 to 1?

Data:

Each datum the rmat minnow sends has a two fields each of type PHISH_UINT64.

Restrictions: none

Related minnows: none

NOTE: give citation for RMAT

slowdown minnow

Syntax:

```
slowdown delta
```

- delta = delay in seconds

Examples:

```
count
```

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 does not care what kinds of datums it receives and re-sends.

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 pairts 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.

NOTE: would really be OK if received any kind of int? Is there some way to check for this, so code doesn't have to be too specific?

NOTE: Could type of its 2nd value be ignored? Is a C++ vector issue, might also need way to copy single arbitrary field from recv buffer to input buffer

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"
```

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 any string with 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 -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 any string with 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"
```

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 any string with 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 in 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