PHISH Package Users Manual

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

Table of Contents

PHISH 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. The version date is printed to the screen every time you process an input script with the bait.py tool. It is also in the file src/version.h 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.

The PHISH package is simple, portable framework that allows you to hook together a collection of independent stand-alone programs so that they can solve a problem together by sending messages to each other. Often this involves processing large volumes of data that arrives in a continuous, streaming fashion or is read from a large archive of files; hence the term "informatics" in the acronym.

The individual programs can be written in any language (e.g. C, C++, Fortran, Python) and link to the PHISH library so that they can receive incoming data from upstream sources, bundle their data, and send it downstream.

A PHISH input script allows specification of the set of programs to launch as independent procsses, how many copies of each are inovked, and the topology of how they connect to each other as they send and receive data. Thus the input script encodes an algorithm for processing continuous data from a stream source or archived data from files. The supported communication topologies include a "hashed" mode of messaging where a "key" is hashed to determine what process to send data to, which is a means of exploiting parallelism in a streaming context, akin to a MapReduce operation.

The sending and receiving of datums between processes is handled within the PHISH library either by calls to the message-passing interface (MPI) library, or by sockets via the ZMQ library. A PHISH tool converts the input script into a configuration file that can be launched by either MPI or a shell script. In the former case, a PHISH program can run on any platform that supports MPI. In the latter case, a PHISH program can run on a single desktop machine or across any network of (geographically distributed) machines that support socket connections.

The PHISH package is open-source software that implements the MapReduce operation popularized by Google on top of standard MPI message passing.

The library is designed for parallel execution on distributed-memory platforms, but will also operate on a single processor. It requires no additional software to build and run, except linking with an MPI library if you wish to perform MapReduces in parallel. Similar to the original Google design, a user performs a MapReduce by writing a small program that invokes the library. The user typically provides two application-specific functions, a "map()" and a "reduce()", that are called back from the library when a MapReduce operation is executed. "Map()" and "reduce()" are serial functions, meaning they are invoked independently on individual processors on portions of your data when performing a MapReduce operation in parallel.

The MR-MPI library is written in C++ and is callable from hi-level langauges such as C++, C, Fortran. A Python wrapper is also included, so MapReduce programs can be written in Python, including map() and reduce() user callback methods. A hi-level scripting interface to the MR-MPI library, called OINK, is also included which can be used to develop and chain MapReduce algorithms together in scripts with commands that simplify data management tasks. OINK has its own manual and doc pages.

The goal of the MR-MPI library is to provide a simple and portable interface for users to create their own MapReduce programs, which can then be run on any desktop or large parallel machine using MPI. See the Background section for features and limitations of this implementation.

The distrubution includes a few examples of simple programs that illustrate the use of MR-MPI.

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

The authors of PHISH are Steve Plimpton and Tim Shead who can be contacted via email: sjplimp,tshead at sandia.gov.

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.

Once you are familiar with PHISH, you may want to bookmark this page at interface_c++.html, since they gives quick access to documentation for all the PHISH library methods and minnows.

PDF file of the entire manual, generated by htmldoc

PHISH has 2 parts: Python script (bait.py) and library (libphish.a)

what is streaming data continuous high bandwidth cannot afford to see it more than once

But can also process corpus of files.

Break stream up into datums = chunk of bytes.

Minnows read (swim in) stream, process datums, can store state, consume or write (altered) datums.

Can wrap existing apps in PHISH (fish-wrapper). Minnows are provided

School of minnows and their connectivity specified in simple input script, processed with bait.py to turn into launchable parallel job, either via MPI and mpirun, or via sockets and a shell script.

easy path to distributed memory parallel for processing streaming data

minnow = stand-alone program (app) that (typically) does one task, designed to work in tandem with other minnows, either copies of itself, or totally different

portable open-source lib, lightweight, easy to call from any language, including Python

each stand-alone minnow calls library

difference between normal datum and "done" datum

explain acronym what each word means

minnow school input script bait.py

school of minnows swim in a stream, doing something coordinated.

This describes the bait.py Python tool that pre-processes a PHISH input script into a form that can be executed by launching an MPI or socket-based job.

Launch syntax:

bait.py -switch values ... < in.script

Command-line args for -switch and their values are discussed below. The in.script is a PHISH input script (text file) that contains a list of Bait.py input commands.

Command-line arguments

-np Np Np = process count default = 1 -var ID str1 str2 ... assign strings to a variable ID default = no variables defined -hostfile filename file with list of processors to run on default = "" -configfile filename filename = MPI config file to create default = "configfile" -path path1:path2:path3:... colon-seperated paths to prepend to app executable names default = "." -mode style manner in which PHISH job will be launched style = "mpich" or "openmpi" or "socket" default = "mpich"

each switch has an allowed abbreviation

-np or -n -var or -v -hostflie or -h -configfile or -c -path or -p -mode or -m

---

Input Script syntax

parsing rules

blank lines OK comment char = #, delete all trailing continuation char = &, with nothing following (including spaces)

variables are as follows ...

variable substitution:

what are rules for this? should probably be $a for single-char variable, $*abc* for multichar

A minnow is an app, which is ...

This section covers these topics:

- Structure of a minnow
- How a minnow shuts down
- Debugging a minnow
- Special issues to be aware of

This is list of minnows provided with PHISH. Some are provided in C++, some in Python, some in both.

- count
- file2fields
- file2words
- filegen
- ping
- pong
- print
- readgraph
- rmat
- slowdown
- sort
- wrapsink
- wrapsource
- wrapss

These are stand-alone apps that can be wrapped as a child process:

- echo
- reverse

Structure of a minnow

command line args setup calls to lib setting callbacks

loop vs probe vs recv

receving and unpacking a datum

packing and sending a datum

How a minnow shuts down

one-way schools, triggered by head done messages, per port level calls to exit, close, etc special care for ring or chain connections or when school has loops Ctrl-C option output at end

---

Special issues to be aware of

shutdown in ZMQ is buggy

This is the API to the PHISH library that minnows (stand-alone applications) call. This is a C-style API, so it is easy to write minnows in any language, e.g. C, C++, Fortran, Python, that call the PHISH library.

Decide where to doc the Python interface (in each doc page)? Is the Sockets interface identical?

PHISH minnows send and receive datums by communicating with other minnows. Before looking at the specific library calls for sending and receiving datums, its helpful to understand how the data in a datum is structured by the PHISH library.

This is discussed below, in the sub-section What is a datum?.

The PHISH library is not large; there are only a handful of calls. They can be grouped into the following 5 categories. Follow the links to see a doc page for each library call. A general discussion of how minnows call these functions is given in the Minnows section of the manual.

1. Library calls for initialization
   phish_init()
   phish_init_python()
   phish_input()
   phish_output()
   phish_done()
   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()
   phish_pack_datum()
   phish_pack_raw()
   phish_pack_byte()
   phish_pack_int()
   phish_pack_uint64()
   phish_pack_double()
   phish_pack_string()
   phish_pack_int_array()
   phish_pack_uint64_array()
   phish_pack_double_array()
5. Miscellaneous library calls
   phish_world()
   phish_reset_receiver()
   phish_error()
   phish_warn()
   phish_timer()

Discuss limits and options affecting library:

make MAXBUF and PHISH_SAFE_SEND be settable options by user?

What is a datum?

A datum is a chunk of bytes sent from one PHISH minnow to another. This section describes the format of the chunk, which is the same whether the datum is sent via MPI or via sockets.

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

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

- #define PHISH_RAW 0
- #define PHISH_BYTE 1
- #define PHISH_INT 2
- #define PHISH_UINT64 3
- #define PHISH_DOUBLE 4
- #define PHISH_STRING 5
- #define PHISH_INT_ARRAY 6
- #define PHISH_UINT64_ARRAY 7
- #define PHISH_DOUBLE_ARRAY 8

PHISH_RAW is a list of raw bytes, which can be of any length, and which the user can format in any manner. PHISH_BYTE is a single byte. PHISH_STRING is a NULL-terminated C-style string. The NULL is included in the datum. The ARRAYS are contiguous lists of int, uint64 or double values.

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

The field data is packed into the datum in a contiguous manner.

NOTE: what about alignment with mix of int/double, or when single PHISH_BYTE are packed?

The maximum allowed size of a datum (in bytes) is set by MAXBUF in src/phish.cpp, which defaults to 1 Mbyte.

Additionally, MPI flags the messages with a "tag". This tag encodes the receiver's port and also a "done" flag. If it is not a done message, the tag is the receiver's 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.

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

MPI also allows the receiver to query the byte size of the message, which is used for the "full" pack/unpack calls below.

This is the list of example PHISH input scripts provided:

** in.pp = ping-pong test between 2 processes

2 procs, each connect to each other via one2one send message of M bytes back-and-forth N times

** in.test = test bait.py syntax processing

arbitrary commands to test that bait.py can process it correctly

** in.wc = word count from files

open files from list convert to words count word occurrence sort to keep top N print the results

can use many procs for file reading and accumulating counts like MapReduce

** in.wrapsink = reverse each filename in a list of filenames

uses wrapssink pass filenames to simple standalone reverse program can launch multiple instances of reverse

** in.wrapss = reverse each filename in a list of filenames

uses wrapss wrap simple standalone reverse program can launch multiple instances of reverse

How to wrap the PHISH lib with Python.

This allows minnows to be written in Python and make calls to the PHISH library.

Include setup of Python info from MR-MPI.

def init(arguments=sys.argv): """Initializes phish and returns a context object for communicating with the rest of the school. Takes the process argv (which may be modified) as its sole argument."""

class context: """Abstract interface for a Phish context, which is used for communication by individual minnows."""

def name(self): """Returns the human-readable name for this minnow."""

def input_ports(self): """Returns a list of connected input ports. Note that this list is kept up-to-date as input ports are closed."""

def output_ports(self): """Returns a list of connected output ports. Note that this list is kept up-to-date as output ports are closed."""

def enable_input_port(self, port, message_callback, closed_callback=None, optional=False): """Specifies an input port to be enabled, along with a callback to be called when a message is received on the port, an optional callback to be called when the port is closed (i.e. all connections to the port are closed), and a flag specifying whether the port is optional (i.e. whether it is an error condition if there are no connections to this port)."""

def enable_output_port(self, port): """Specifies an output port to be enabled. It is an error to attempt sending a message on a port that hasn't been enabled, or to make a connection to such a port."""

def set_last_port_closed_callback(self, callback): """Specifies a callback to be called exactly once when every input port has been closed. Note: this callback will never be called for minnows that have no input ports."""

def loop(self): """Called to begin an event loop that will receive messages and invoke registered callbacks. Loops cannot be nested, so subsequent calls to loop() before calling loop_complete() will be ignored."""

def loop_complete(self): """Called to exit an event loop started with loop(). Note that multiple calls to loop_complete() and calls to loop_complete() before calling loop() will be ignored."""

def send(self, message, output_port=0): """Sends a message to a single minnow connected to the given output port, choosing recipients in round-robin order."""

def send_all(self, message, output_port=0): """Sends a message to every minnow connected to the given output port."""

def send_hashed(self, key, value, output_port=0): """Sends a key-value pair to a single minnow connected to the given output port, choosing recipients based on a hash of the key."""

def close_port(self, output_port): """Notifies downstream minnows that the given output port has been closed."""

def close(self): """Closes the context and releases any resources allocated by the library. Note that this also implicitly closes any open output ports and completes any running loop."""

def school(backend="zmq"): """Factory function for creating school objects. The backend argument is used to

specify the backend to use for communications. Currently, only "zmq" is supported as a backend."""

class school: """Abstract interface for managing a collection of minnow processes."""

def create_minnows(self, name, arguments, count=1, hosts=None): """Starts one-or-more minnow processes, using the given command-line arguments, count of processes to create, and optional list of hosts where processes should be run, returning a list containing handles to the new minnows. The hosts parameter must either be a single host, in which case all the new minnow processes will be run on that host (useful to put I/O minnows close to disk hardware, for example); or, the hosts parameter must be a list of host names, one-per-minnow."""

def all_to_all(self, output_minnows, output_port, input_port, input_minnows): """Creates all-to-all connections from the output ports on one set of minnows to the input ports of another."""

def one_to_one(self, output_minnows, output_port, input_port, input_minnows): """Connects an output port from one minnow to the input port of the corresponding minnow, for two equal sets of minnows."""

def start(self): """Notifies the school that all minnows have been created / connected, and processing may begin."""

* connect ID1:oport style ID2:iport

connect output port (oport) of sp ID1 to input port (iport) of sp ID2

style = kind of connection style = single or paired or hashed or roundrobin or direct or bcast or chain or ring or publish or subscribe single = N senders with a single receiver, N = 1 is OK paired = N senders and N receivers, senders and receivers are paired so that each sender sends to one receiver, N = 1 is OK hashed = N senders and M receivers, each sender hashes datum on a key to decide which of M receivers to send to, N and/or M = 1 is OK roundrobin = N senders and M receivers, each sender sends datums to each of M receivers in round-robin fashion, N and/or M = 1 is OK TIM: direct = N senders and M receivers, each sender sends a datum to a specific receiver it chooses when the message is sent N and/or M = 1 is OK bcast = N senders and M receivers, each sender sends a datum to all M receivers, N and/or M = 1 is OK chain = N procs, N > 1 is required, procs are connected as a linear chain, each sending to one downstream neighbor, 1st proc is not a receiver, last proc is not a sender ring = N procs, N > 1 is required, procs are connected as a circular ring, each sending to one downstream neighbor, every proc is both a sender and receiver TIM: publish = send datums to a socket one or more external programs can read datums from the socket socket is specfied by replacing ID2:port with tcpport tcpport = a number, e.g. 25 this style is only supported by ZMQ, not by MPI NOTE: anyway around this restriction? let MPI lib write to a ZMQ socket? TIM: subscribe = receive datums from a socket one or more external programs can write datums to the socket socket is specfied by replacing ID1:port with host:tcpport host = hostname, e.g. www.foo.com tcpport = a number, e.g. 25 this style is only supported by ZMQ, not by MPI NOTE: anyway around this restriction? let MPI lib read from a ZMQ socket?

for chain and ring, ID1 = ID2 is required

iport and oport are optional (no colon if not specified) if not specified, port 0 is used

each app defines how many input and output ports it recognizes if an app support M ports, they are numbered 0,1,...,M-1

an app's input port can be connected to multiple outputs by appearing in multiple connect commands

an app's output port can be connected to multiple inputs by appearing in multiple connect commands

* layout ID nprocesses keyword value ...

ID = minnow ID nprocesses = # of processes to launch for each minnow if layout not specified for a minnow, nprocesses = 1 is default zero or more keyword/value pairs can be appended possible keywords = host or prepend host value = machine name, used for sockets need to define this syntax more clearly prepend value = name of executable to launch minnow under in config file examples = python or /usr/local/bin/python2.7 or valgrind

* minnow ID exe args

define a minnow with ID exe = name of stand-alone executable args = args passed to executable when launched

exe is looked for in list of paths

* set keyword value

could allow max buf size for a datum to be set then append switch to all launched minnows nothing defined yet could duplicate command-line switches to allow them to be set in input script could also allow verbosity or makefile verbosity = 0 or 1 (default = 0) = currently not used makefile = filename (default = Makefile) = currently not used should be used to build EXE files

* variable ID str1 str2 ...

assign one or more strings to a variable ID

count instances of keyed datums

syntax: no args

creates an internal hash, so it can count instances of keys hash key = string, hash value = count keeps track of largest key it receives

for each datum: treat buf of nbytes as a string and hash it increment hash value for that string

when done: iterate over hash table send all hash entries downstream as count/string

echo lines from stdin to stdout

file2fields minnow

read file and emit words

syntax: no args

for each datum: treat message as filename open it, parse into words separated by whitespace send each word downstream, using word as key

emit filenames

syntax: filengen file1 file2 ...

filenames can also be directories directories are opened recursively and their files added to list send each file name one by one downsteram

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

* void phish_done(void (*done)())

call once for each input port the app uses cannot call after phish_check()

iport = port ID (0 to MAXPORT-1) datumfunc = app function that will be called when a datum is recv by port, datumfunc(is passed the number of fields in the received datum datumfunc can be NULL if don't wish to process datums donefunc = app function that will be called when port is closed by receiving DONE messages from all senders donefunc can be NULL reqflag = 1 if input port must be used by PHISH script, else 0

error if input script uses input port not setup by phish_input() error if input script does not use a required port OK if input script does not use a non-required port

call once for each output port the app uses cannot call after phish_check()

iport = port ID (0 to MAXPORT-1) all output ports are non-required OK if input script does not use a port

set callback function done() to invoke when all input ports are closed

* void phish_check()

checks that input/output ports defined by app are consistent with usage in PHISH input script call after phish_input() and phish_output() are called for all ports

changes all CLOSED output ports requested by input script to OPEN cannot be done in phish_output() since may not be called

* void phish_close(int iport)

close an output port with portID iport sends DONE message to all downstream receivers no-op if port is already closed

* int phish_datum(char **buf, int *len)

return info for entire received datum, including its field delimiters buf = ptr to start of datum len = byte length of entire datum, including field delimiters also return input port message was received on

Using this call allows the minnow to do whatever it wishes with the datum, including send it along intact to another minnow. See the phish_pack_datum() function.

This call does not conflict with the phish_unpack functions that unpack the datum, field by field. Those can still be used to process the same datum.

* void phish_error(char *str) * void phish_warn(char *str)

print error string does not close output ports for MPI, calls MPI_Abort()

print warning string

* void phish_exit()

warns if any input ports still open closes all output ports if still unclosed free memory for PHISH library must be last call to any phish function

* void phish_init(int *pnarg, char ***pargs) * int phish_init_python(int *narg, char **args) * int phish_world(int *pme, int *pnprocs)

wrapper on phish_init() to allow calling from Python

1st call that app makes, to initialize PHISH library must be called before any other phish function

pnarg = ptr to narg pargs = ptr to args

looks for args of the following form and strips them off note that -in and -out may appear multiple times in arg list modifies narg and args to only have args that follow -args

-app exe ID N Nprev exe = name of executable file for this app (e.g. count or count.py) ID = ID string of this app in PHISH input script N = how many instances of this app were launched Nprev = how many processes exist preceding this app -in sprocs sfirst sport style rprocs rfirst rport one connection between pair of ports for incoming datums I receive sprocs = # of procs on sender side sfirst = first proc on sender side sport = port ID on sender style = single or paired or hashed or roundrobin or chain or ring rprocs = # of procs on receiver side, including me rfirst = first proc on receiver side rport = port ID I receive on -in sprocs sfirst sport style rprocs rfirst rport one connection between pair of ports for outgoing datums I send sprocs = # of procs on sender side, including me sfirst = first proc on sender side sport = port ID I send on style = single or paired or hashed or roundrobin or chain or ring rprocs = # of procs on receiver side rfirst = first proc on receiver side rport = port ID on receiver -args arg1 arg2 ... = args for the app itself

each specified input port is set to CLOSED instead of UNUSED each specified output port is set to CLOSED instead of UNUSED

return MPI_Comm return ID of this process in MPI sense (me = 0 to nprocs-1) return # of processes in MPI sense (nprocs)

* void phish_pack_datum(char *buf, int n) * void phish_pack_raw(char *buf, int n) * void phish_pack_byte(char cvalue) * void phish_pack_int(int ivalue) * void phish_pack_uint64(uint64_t uvalue) * void phish_pack_double(double dvalue) * void phish_pack_string(char *str) * void phish_pack_int_array(int *ivec, int n) * void phish_pack_uint64_array(uint64_t *uvec, int n) * void phish_pack_double_array(int *dvec, int n)

pack an entire datum of N bytes, including its field delimiters (see phish_datum() function) must be the 1st and only pack call before sending ?? can we check this? pack raw buf of N bytes into send datum pack cvalue into send datum pack ivalue into send datum pack uvalue into send datum pack dvalue into send datum pack str with its trailing NULL into send datum pack N integers in ivec into send datum pack N uint64s in uvec into send datum< pack N doubles in dvec into send datum

* void phish_loop() * void phish_probe(void (*probe)()) * int phish_recv()

continuous loop, waiting for input datums if datum arrives on an input port, the port process() is called if port is closed, the port done() is called when all ports are closed, calls all_done callback, exits

same as phish_loop(), except that incoming datums are probed for in non-blocking manner if no message is available, then probe() function in invoked so app can do more work

probe for incoming message, return whether one available or not allows app to request messages explicitly, rather than thru phish_loop/probe and a callback return 0 if no message return 1 if done message (and invokes callbacks if necessary) return N for # of fields in received message

* void phish_reset_receiver(int oport, int receiver)

change the process I send messages to on output port oport can be used to effectively permute the ordering of processes in a ring receiver = 0 to M-1, where M = # of receivers in the connection only supported by "ring" connection

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

send a packed datum to output port oport

send a packed datum to output port oport in a hashed manner using key of length nbytes to determine who receiver is only supported by "hashed" connections

send a packed datum to output port oport and to a specific recieving process receiver = 0 to M-1, where M = # of receivers in the connection only supported by "direct" connections

* double phish_timer()

wrapper on MPI_Wtime() returns current time can compute elapsed time between two phish_timer() calls

* int phish_unpack(char **buf, int *len)

return next field in the received datum return value of field = PHISH_RAW, PHISH_INT, etc buf = ptr to start of field data len = # of values in field for PHISH_RAW, len = # of bytes for PHISH_STRING, len = # of bytes including NULL for PHISH_BYTE, PHISH_INT, PHISH_UINT64, PHISH_DOUBLE, len = 1 for PHISH_INT_ARRAY, PHISH_UINT64_ARRAY, PHISH_DOUBLE_ARRAY, len = # of array values

NOTE: These only return a ptr to the field data, so if you want the data to persist when the next datum is received by the minnow (e.g. during a phish_loop() or phish_probe()) then you must make a copy. It is OK to unpack several fields from the same datum before making copies of the fields.

reflect messages to a receiver

syntax: ping N M N = # of times to ping/pong with partner proc M = # of bytes in a message

fill M-byte buffer with NULLs send it and go into loop

for each datum: when recv from partner increment count and send back to partner when count hits M, send done message

reflect messages to a sender

syntax: no args

datum method: when recv from partner, send message back to partner

print datums to screen or file

syntax: print -f filename -f is optional, if not specified, prints to stdout

for each datum: print string

when done: close file

readgraph minnow

reverse characters in lines from stdin and write to stdout

rmat gen minnow

read datum and emit it with slowdown delay

syntax: slowdown delta delta = time to delay (in seconds)

read each datum and insure delay seconds have passed before writing it downstream

for each datum: query time since last datum was processed and invoke usleep() if needed send entire datum downstream

sort datums, emit highest count ones

syntax: sort N N = keep top N of sorted list

for each datum: assume message is int/string store in/string as STL pair in a vector list

when done: sort the list based on integer count send the top N list items downstream as count/string

wrap a child process which consumes datums by reading from stdin

syntax: wrapsink "program" "program" can be any string with flags, redirection, etc enclose in quotes to prevent shell from processing it

write datums to child, one by one, as lines of input write done via popen pipe

for each datum: write datum to pipe with appended newline

when done: close the pipe

wrap a child process which creates datums by writing to stdout

syntax: wrapsource -f "program" -f is optional

if -f is specified, receive filenames in stream and invoke child process on each filename generate "program" via sprintf() using filename as arg so "program" presumably has %s in it if -f is not specified, invoke child process just once using "program"

"program" can be any string with flags, redirection, etc enclose in quotes to prevent shell from processing it

read lines of output from child one by one as datums via a pipe send them downstream

for each datum: launch the child process on the filename read all its output until child exits send each line of output downstream vis phish_send w/out newline

wrap a child process which both consumes and creates datums via stdin/stdout so child is a sink and a source

syntax: wrapss "program" "program" can be any string with flags, redirection, etc enclose in quotes to prevent shell from processing it

open 2 pipes to child via pipe() fork() into parent and child processes parent calls phish_probe() to query incoming messages and child output child hooks its stdin/stdout to 2 pipes via dup2() child invokes the "program" via execv()

datum method: write datum to pipe with appended newline

probe method: poll pipe for output from child if output is there, read it and break into lines phish_send() each line downstream as string w/out newline

close method: close write pipe to child so it will know parent is done wait for all output from child read pipe send DONE message to notify receivers