CSCI 5253 Final Project Report

Zane Jakobs

December 10, 2020

1 Overview And Goals

The main purpose of this project, which is available at `https://github.com/DiffeoInvariant/datacenter-profiling`, is to provide an extensible framework for the use of Extended Berkeley Packet Filter (eBPF)-based network profiling in MPI clusters. If a programmer goes to the effort of writing and debugging an MPI program—or even if the user simply wants to compare among existing programs—they will often care about that program running fast and efficiently on a cluster. Often, network bandwidth is the biggest bottleneck of such programs, particularly in scientific computing, so a programmer interested in a high-performance application will want to be able to monitor their application's network usage, ideally with minimal CPU and network overhead. Libraries like PETSc[1] enable such tracking easily — for instance, any PETSc program can be run with the command-line option $-\text{log\_view}$ to view a summary of the program's performance statistics, including network usage. However, MPI programmers that are not using PETSc or a similar library generally will have to recompile their application to get a network usage profile or trace from tools like VampirTrace or PMPI. As anyone who has had to compile a large scientific application knows, this is not always as straightforward as it sounds, and can take a long time even when the process goes smoothly.

One solution to this problem comes from the modern linux kernel, which enables very low-level tracing of both kernel- and user-space events by attaching eBPF programs to kprobes. [2] The BPF Compiler Collection (BCC)[3] provides a set of pre-written eBPF programs that are useful for instrumentation of all sorts, including network profiling. In particular, the programs

[1] `https://www.mcs.anl.gov/petsc/`

[2] `https://lkml.org/lkml/2015/4/14/232`

[3] https://github.com/iovisor/bcc

tcpaccept(8), tcpconnect(8), tcpconnlat(8), tcplife (8), and tcpretrans(8) can be combined to produce a useful summary of a program's network usage.

Specifically, if each node in an MPI cluster is running all of the above-named eBPF programs (tcpaccept(8) and friends), their outputs can be combined together to produce process-wise summaries of TCP traffic, which we store in a hash table whose keys are process ID numbers (PIDs). Periodically, each node pushes its summaries into an MPI-backed message queue, and those queues are sent back to the root node. After receiving those messages containing summaries on other MPI ranks (each running on its own node), the root node writes them to an output file. Additionally, as soon as data is available, the root node launches a Flask server that serves requests to a REST API that allows the user to request all collected data, all data on a certain MPI rank with a given PID, or all data for all processes (running on any MPI rank) with a given name.

2 Description of Components

The project can be broadly subdivided into 7 components: the eBPF programs, data structures to serialize and efficiently encode their output and summaries of it, an MPI-backed message queue, a REST API and server, the driver programs, an optional containerized version of the program (hosted on Singularity hub) and for testing purposes, cluster setup and installation scripts to build an SDN, specifically a LAN. Since the project is, with the exception of the REST server, written in C, and the PETSc library provides many of the utilities programmers expect from "modern" languages that C doesn't provide, as well as an easy interface to install many useful C libraries through its configure script (and update the relevant environment variables to use those libraries in a makefile), we make heavy use of PETSc in our code. For example, we use a hash map in the form of a PETSc wrapper around a KHash hash map, and our message queue stores PetscBag instances, which can in turn store any of the data and summary structs we define.

eBPF Programs

Due to time constraints, we used only the five BCC-provided programs mentioned in the first section: tcpaccept(8), tcpconnect(8), tcpconnlat(8), tcplife (8), and tcpretrans(8). Through a script named collect −tcp−data.sh that must, at present, be run as root[4], each of these programs is run in the background with its output redirected to an appropriately named file. This must be done on every node in the cluster before running the program to be profiled; this can be accomplished with OpenMPI on N nodes by running

```
sudo mpirun -np N --map-by ppr:1:node --allow-run-as-root ./collect-tcp-data.sh &
```

from the root node. Every MPI implementation I've seen will warn you that you should not run as root or outright disallows it by default. In OpenMPI, this can be circumvented with the $--allow-run-as-root$ option. The requirement that the data collection script be run as root presents the most significant limitation of this software: it must be run on a cluster to which the user has root access.

We also define a C struct for each program that can hold the information in one line of output, as well as a C struct to hold data from all the programs collected about an individual process, and one to hold a summary of that process's data.

Serialization

To allow the user to extend the program to collect more sophisticated summaries, each of the above-mentioned C structs can be sent in an MPI message after the user has called the register_mpi_types() function. This was accomplished with MPI's MPI_Type_create_struct() and MPI_Type_commit() functions. By default, the output summaries are stored in plain ASCII text in a file. However, if the user has extremely limited disk space and wants to use a more efficient encoding, then as mentioned above, each of those structs can be serialized in a PetscBag, which is a C class provided by PETSc that can be used for, among other things, efficiently serializing data in a binary format. Additionally, for testing purposes we wrote a Python script to download matrices from the Matrix Market (`math.nist.gov/MatrixMarket`) and serialize them into the space-efficient PETSc binary format.

The Message Queue

Our extremely simple message queue is built based on a circular buffer and the MPI_Reduce() and MPI_Gather() functions. We decided that it would be easier, both for development but crucially for the end user, to implement a simple message queue with MPI than to integrate a "standard" message

[4] Since all eBPF programs must be run as root, although this may change in the future. [CITE Ingo Molnar `https://lkml.org/lkml/2015/4/14/232`]

queue like RabbitMQ or ZeroMQ into our cluster and program setup. Its operation consists of all nodes (possibly including the root node) collecting data and pushing it into their respective buffers, then gathering that data to the root node, clearing the non-root nodes' buffers. Since this operation requires the use of MPI_Reduce() (to figure out how many items each process is sending so that the root node can allocate enough memory to store them) and MPI_Gather() (to perform the actual operation), both of which are blocking[5], the user does not need to worry about using MPI_Barrier() to synchronize the communication. This is implemented in the buffer_gather() function in our code.

Driver Programs

There are two C programs that run the code contained in petsc_webserver.h, petsc_webserver.c, and petsc_webserver_backend.py: petsc_webserver_driver.c and webserver_launcher.c. The main driver first reads through all the relevant input files and stores the data it reads. It then uses MPI_Comm_spawn() to spawn a subprocess on the root node (running webserver_launcher.c) that launches the REST server via the Python C API (specifically, with PyRun_SimpleFile()). The driver then continues into an infinite loop, where it polls the data files to check if they have new data, and if they do, reads that new data, updates the summaries, then overwrites the old output file with the updated output. The driver program can be run with the following options, produced by running it with the -h (or -help) flag (excluding options common to all PETSc programs, which are also printed out):

```
PETSc webserver: This program periodically reads the output of the eBPF programs
tcpaccept, tcpconnect, tcpconnlat, tcplife, and tcpretrans, summarizes that data, and stores that
data in a message queue. The summaries are then gathered to the root node (MPI rank 0), where it is
written to an output file. Once the existing input files have been read to their ends, the root
process spawns a new subcommunicator via MPI_Comm_spawn() and launches a Python webserver
(currently written with Flask) on that spawned process that reads the output file, and serves reques
to a REST API. Available endpoints are:
-------------------------------------------------------------------------------------------
GET /api/get/all (gets data for all processes on all MPI ranks)
GET /api/get/{rank}/{pid} (gets data for the process with PID {pid} on MPI rank {rank})
GET /api/get/{name} (gets data on all MPI ranks for all processes with name {name}
-------------------------------------------------------------------------------------------
```

[5] i.e. they block all processes in the communicator until every other process has finished executing the function

```
Usage:
Options:
--python_server [filename] : (optional, default petsc_webserver_frontend.py) filename of Python web
       you want to launch
--python_launcher [filename] : (optional, default ./webserver_launcher) filename of the MPI program
        that can be launched as an MPI child with MPI_Comm_split() with argv
        'python3 <python_server> -f <output> -p <port>' and will run a webserver on the
         appropriate port.
-file [filename] : (optional if any of --XXX_file are given) input file
-type [TCPACCEPT,TCPCONNECT,TCPRETRANS,TCPLIFE,TCPCONNLAT] : (required only if -file is given) what
       sort of input file is the -file argument?
--accept_file [filename] : (optional) file for tcpaccept data
--connect_file [filename] : (optional) file for tcpconnect data
--connlat_file [filename] : (optional) file for tcpconnlat data
--life_file [filename] : (optional) file for tcplife data
--retrans_file [filename] : (optional) file for tcpretrans data
-o (--output) [filename] : (optional, default stdout) file to output data to
-p (--port) [port (int)] : (optional, default 5000) which TCP port to use to serve requests
--buffer_capcity [capacity] : (optional, default 10,000) size of the buffer (number of entries)
--polling_interval [interval] : (optional, default 5.0) how many seconds to wait before
       checking the file for more data after reaching the end?
```

The root endpoint displays HTML representing all the data we have so far, and the endpoint /<name> retreives HTML representing all entries for the given program name. Note that the driver MUST be started after TCP data collection has started in order to guarantee that every file the driver reads in exists.

I chose this design because I want this software to be as portable as it can be across different clusters, so I tried to rely as little as possible on anything outside of the C standard, POSIX (since eBPF requires a modern linux kernel anyway), the MPI standard or PETSc. However, one notable drawback is that I can't do online updates of the server frontend because it's launched from MPI_Comm_spawn within the backend, so updating the frontend requires restarting the backend (which automatically launches the frontend). It would be possible to make this happen automatically by providing a git hook to install on the server so that on each commit, git rebuilds the server if the backend changes and restarts it if either the backend or frontend changes, but we did not do this.

REST API

The Python program launched in the drivers above takes in two command-line options, one (-f) being the file containing its input data (the output of the driver), the other (-p) being the TCP port that Flask should use. The endpoints for the REST API are shown in the help message above. Upon receiving a request, the server re-reads the input data (in case it has been updated since the last request), constructs the appropriate response, and responds. I originally intended to use SAWs (Scientific Application Webserver), a library that PETSc integrates with that would have allowed me to bypass the output file and subprocess running the server entirely. However, the PETSc developers haven't written the code necessary to use PetscBag objects with SAWs (which is what I was planning to do), so for the sake of time, since implementing even a very simple function for a library like PETSc can be a significant time investment, we decided to use a Flask-based server for now. In case the user doesn't feel like writing their own REST calls (even though this API is extremely simple, and requests can easily be constructed manually), we have provided a Python-based web client.

Singularity Container

In case the user doesn't want to build the software on their cluster, we have provided the software in the form of a Singularity container at [INSERT SHUB LINK]. Also, if it ever becomes the case that eBPF programs can be run without root privileges, this container will allow the software to be run on clusters whose users do not have root access. For ease of use on systems that use Modules, we also provide a Tool Command Language (TCL) modulefile to provide access to Singularity.

Cluster Setup Scripts and SD-LAN

In addition to providing a containerized version of the software, we wrote some shell scripts to install the relevant dependencies and software on an Ubuntu VM. To test the software, we set up two VMs in Chameleon Cloud running Ubuntu 20.04, and, following the guide at `https://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/`, set up a LAN between the two with ssh-agent, with a distributed filesystem running NFS that we used to retreive debugging information from the non-root node, as well as distributing the executable among the nodes.

3 Capabilities and Limitations

This project provides a low-overhead[6] option to profile the network usage of an MPI program and transmit the results to a remote user via a REST API (including browser-friendly HTML). It does not require the user to recompile their code, only to run the data collection programs and then the server before running their program. This could be done explicitly on the command-line or in a batch script, running the data collection programs and server in the background. Furthermore, even an inexperienced user should be able to extend this code to collect more detailed data if so desired. However, because eBPF programs can currently only be run as root, the user needs root access to use this software in its current form, which renders it unusable on many HPC clusters. If one day eBPF programs can be run as a non-root user, this software will be able to be run on essentially arbitrary clusters via the provided Singularity container. Currently, the provided container works in case the user doesn't want to install the relevant dependencies on their machine (which can be done automatically on debian-based systems with a script we wrote, ubuntu−install.sh).

Appendix A

This code — available at `https://github.com/DiffeoInvariant/datacenter-profiling` — defines the public API of the mini-library that powers the driver program, itself available along with the implementation of this header and the Python webserver code in that git repository.

```
#ifndef DCPROF_PETSC_WEBSERVER_H
#define DCPROF_PETSC_WEBSERVER_H
#include <petsc.h>
#include <unistd.h>
#include <petsc/private/hashtable.h>
#include <petsc/private/hashmap.h>

#define IP_ADDR_MAX_LEN 45
#define COMM_MAX_LEN    PETSC_MAX_PATH_LEN

typedef enum {TCPACCEPT,TCPCONNECT,TCPCONNLAT,TCPLIFE,TCPRETRANS} InputType;
static const char *InputTypes[] = {"ACCEPT","CONNECT","CONNLAT","LIFE","RETRANS","
    TCP",0};

typedef enum
```

[6] Even if the data collecting programs have generated over 20,000 total entries, the program uses less than 500 kB of network data (i.e. transmits ¡ 500 kB over the network).

```
  {
   DTYPE_ACCEPT=0,
   DTYPE_CONNECT=1,
   DTYPE_CONNLAT=2,
   DTYPE_LIFE=3,
   DTYPE_RETRANS=4,
   DTYPE_SUMMARY=5
  } SERVER_MPI_DTYPE;

MPI_Datatype MPI_DTYPES[6];

/* call this at the beginning of the program, but after MPI_Init(), for
   any program that needs to send any of the XXX_entry types here, as well
   as process_data_summary. The registered types are stored in MPI_DTYPES
   and indexed by the SERVER_MPI_DTYPES enum. */
extern PetscErrorCode register_mpi_types();

typedef struct {
  PetscInt pid, ip, rport, lport;
  char     laddr[IP_ADDR_MAX_LEN],raddr[IP_ADDR_MAX_LEN],comm[COMM_MAX_LEN];
} tcpaccept_entry;

/* creates a PetscBag with PetscBagCreate() to serialize a
   tcpaccept_entry, and stores  a pointer to that entry
   (via PetscBagGetData()) in the first parameter. The second parmeter
   is used to store the PetscBag we create. The third parameter indicates
   how many entries you have created already to ensure that each entry has
   a unique name, even if data like the PID and/or process name are the
   same. */
extern PetscErrorCode create_tcpaccept_entry_bag(tcpaccept_entry **, PetscBag *, PetscInt);

/* parses a line of the form
   PID COMM IP RADDR RPORT LADDR LPORT
   which is stored in the second parameter,
   and the result is written to the first parameter */
extern PetscErrorCode tcpaccept_entry_parse_line(tcpaccept_entry *, char *);

typedef struct {
  PetscInt pid,ip,dport;
  char     saddr[IP_ADDR_MAX_LEN],daddr[IP_ADDR_MAX_LEN],comm[COMM_MAX_LEN];
} tcpconnect_entry;

/* creates a PetscBag with PetscBagCreate() to serialize a
   tcpconnect_entry, and stores  a pointer to that entry
   (via PetscBagGetData()) in the first parameter. The second parmeter
   is used to store the PetscBag we create. The third parameter indicates
   how many entries you have created already to ensure that each entry has
   a unique name, even if data like the PID and/or process name are the
```

```c
   same. */
extern PetscErrorCode create_tcpconnect_entry_bag(tcpconnect_entry **, PetscBag *, PetscInt);

/* parses a line of the form
   PID COMM IP SADDR DADDR DPORT
   which is stored in the second parameter,
   and the result is written to the first parameter */
extern PetscErrorCode tcpconnect_entry_parse_line(tcpconnect_entry *, char *);

typedef struct {
  PetscInt  pid,ip,dport;
  PetscReal lat_ms;
  char      saddr[IP_ADDR_MAX_LEN],daddr[IP_ADDR_MAX_LEN],comm[COMM_MAX_LEN];
} tcpconnlat_entry;

/* creates a PetscBag with PetscBagCreate() to serialize a
   tcpconnlat_entry, and stores  a pointer to that entry
   (via PetscBagGetData()) in the first parameter. The second parmeter
   is used to store the PetscBag we create. The third parameter indicates
   how many entries you have created already to ensure that each entry has
   a unique name, even if data like the PID and/or process name are the
   same. */
extern PetscErrorCode create_tcpconnlat_entry_bag(tcpconnlat_entry **, PetscBag *, PetscInt);

/* parses a line of the form
   PID COMM IP SADDR DADDR DPORT LAT(ms)
   which is stored in the second parameter,
   and the result is written to the first parameter */
extern PetscErrorCode tcpconnlat_entry_parse_line(tcpconnlat_entry *, char *);
#define TIME_LEN 9

typedef struct {
  PetscInt  pid,ip,lport,rport,tx_kb,rx_kb;
  PetscReal ms;
  char      laddr[IP_ADDR_MAX_LEN],raddr[IP_ADDR_MAX_LEN],comm[COMM_MAX_LEN],time[TIME_LEN];
} tcplife_entry;

/* creates a PetscBag with PetscBagCreate() to serialize a
   tcplife_entry, and stores  a pointer to that entry
   (via PetscBagGetData()) in the first parameter. The second parmeter
   is used to store the PetscBag we create. The third parameter indicates
   how many entries you have created already to ensure that each entry has
   a unique name, even if data like the PID and/or process name are the
   same. */
extern PetscErrorCode create_tcplife_entry_bag(tcplife_entry **, PetscBag *, PetscInt);
```

```c

/* parses a line of the form
   PID,COMM,IP,LADDR,LPORT,RADDR,RPORT,TX_KB,RX_KB,MS
   which is stored in the second parameter,
   and the result is written to the first parameter.
   NOTE: entries of this form can be generated by
   running ‘tcplife -s > /path/to/tcplife.data.file‘,
   since they are comma-delimited, unlike the other
   XXX_entry_parse_line() functions declared in this file
   whose input is space-delimited. */
extern PetscErrorCode tcplife_entry_parse_line(tcplife_entry *, char *);


typedef struct {
  PetscInt pid,ip;
  char     laddr_port[COMM_MAX_LEN],raddr_port[COMM_MAX_LEN],state[COMM_MAX_LEN];/* TODO: find out how long these really should be, cuz this is longer than necessary. not too important though. */
} tcpretrans_entry;

/* creates a PetscBag with PetscBagCreate() to serialize a
   tcpretrans_entry, and stores  a pointer to that entry
   (via PetscBagGetData()) in the first parameter. The second parmeter
   is used to store the PetscBag we create. The third parameter indicates
   how many entries you have created already to ensure that each entry has
   a unique name, even if data like the PID and/or process name are the
   same. */
extern PetscErrorCode create_tcpretrans_entry_bag(tcpretrans_entry **, PetscBag *, PetscInt);


extern PetscErrorCode tcpretrans_entry_parse_line(tcpretrans_entry *, char *);


typedef struct {
  /* a circular buffer */
  PetscBag   *buf; /* array of items */
  size_t capacity,valid_start,valid_end,num_items;
} entry_buffer;

/* creates a circular buffer that can be used as a message queue,
   with MPI for message passing. All MPI ranks can read data and store
   it in the buffer. While you can store any type in the buffer that can
   be serialized in a PetscBag, and on any one rank you can in principle
   store multiple types in the same buffer, you CANNOT store multiple types
   in the same buffer and then call buffer_gather() or buffer_gather_summaries()
   on that buffer, as that invokes undefined behavior in MPI_Gather()

   The first parameter is a pointer to the created buffer, and the second
   parameter is the capacity of the buffer.*/
extern PetscErrorCode buffer_create(entry_buffer *, size_t);

/* frees all memory used by the buffer, calls PetscBagDestroy() on any
   remaining elements */
extern PetscErrorCode buffer_destroy(entry_buffer *);

/* inserts the PetscBag stored in the second parameter in the buffer
   pointed to by the first. Returns 0 on success, non-zero on failure.
   The only possible failure mode is if the buffer is full, in which case
   the insertion does not happen. */
extern PetscInt       buffer_try_insert(entry_buffer *, PetscBag);

/* returns the current number of items in the buffer pointed to by the first
    parameter */
extern size_t         buffer_size(entry_buffer *);

extern size_t         buffer_capacity(entry_buffer *);

extern PetscBool      buffer_full(entry_buffer *);

extern PetscBool      buffer_empty(entry_buffer *);

/* gets the object that has spent the most time in the buffer.
   If you call buffer_get_item() multiple times in a row WITHOUT
   calling buffer_pop() in between each call, you will get the same
   PetscBag on each call to buffer_get_item()*/
extern PetscErrorCode buffer_get_item(entry_buffer *, PetscBag *);

/* remove the object that has spent the most time in the buffer,
   and free its memory */
extern PetscErrorCode buffer_pop(entry_buffer *);

/* gathers all buffer summaries to a buffer on root */
extern PetscErrorCode buffer_gather_summaries(entry_buffer *);

/* gather everything in the buffer pointed to by the first parameter to root,
   with all the entries of the type indicated by the second parameter.
   NOTE: if all the entries are not of that type, that will invoke
   undefined behavior in the MPI_Gather(), and probably cause a crash */
extern PetscErrorCode buffer_gather(entry_buffer *, SERVER_MPI_DTYPE);


typedef struct {
  long long naccept,nconnect,nconnlat,nlife,nretrans,
            tx_kb,rx_kb,nipv4,nipv6;
  PetscReal latms,lifems;
  char      comm[COMM_MAX_LEN];
} process_data;
```

```c
typedef struct {
  PetscInt  pid,rank;
  long      tx_kb,rx_kb,n_event;
  PetscReal avg_latency,avg_lifetime,fraction_ipv6;
  char      comm[COMM_MAX_LEN];
} process_data_summary;

/* write the summary pointed to by the second parameter to the
   file pointed to by the first */
extern PetscErrorCode summary_view(FILE *, process_data_summary *);

/* create a process_data_summary and store it in the third parameter. The
   first parameter is the PID, and the second is a pointer to the
   process_data you want to summarize.*/
extern PetscErrorCode process_data_summarize(PetscInt, process_data *,
    process_data_summary *);

#define pid_hash(pid) pid


#define process_data_equal(lhs,rhs) ( lhs.naccept == rhs.naccept && \
              lhs.nconnect == rhs.nconnect && \
              lhs.nconnlat == rhs.nconnlat && \
              lhs.nlife == rhs.nlife &&       \
              lhs.tx_kb == rhs.tx_kb && \
              lhs.rx_kb == rhs.rx_kb && \
              lhs.nipv4 == rhs.nipv4 && \
              lhs.nipv6 == rhs.nipv6 && \
              lhs.latms == rhs.latms && \
              lhs.lifems == rhs.lifems)

#define int_equal(lhs,rhs) (lhs == rhs)

static process_data default_pdata = {0,0,0,0,0,0,0,0,0,0.0,0.0,"[unknown]"};

PETSC_HASH_MAP(HMapData,PetscInt,process_data,PetscHashInt,int_equal,default_pdata)
    ;

/* sets the values of the process_data to the defaults */
extern PetscErrorCode process_data_initialize(process_data *);

/* calculates what fraction of all TCP events used IPv6 as opposed to IPv4 */
extern PetscReal fraction_ipv6(process_data *);

extern PetscErrorCode create_process_data_bag(process_data **, PetscBag *);

/* the third parameter is the MPI_Comm rank, fourth is the number of the entry*/
```

```
extern PetscErrorCode create_process_summary_bag(process_data_summary **, PetscBag *, PetscInt, PetscInt);

typedef struct {
  PetscHMapData ht;
} process_statistics;

extern PetscErrorCode process_statistics_get_summary(process_statistics *, PetscInt, process_data_summary *);

extern PetscErrorCode process_statistics_num_entries(process_statistics *, PetscInt *);

/* second and third parameters are pointers to array of process_data and PetscInt (pid) respectively, each of the length retrieved from process_statistics_num_entries() */
extern PetscErrorCode process_statistics_get_all(process_statistics *, process_data *, PetscInt *);


extern PetscErrorCode process_statistics_init(process_statistics *);

extern PetscErrorCode process_statistics_destroy(process_statistics *);

extern PetscErrorCode process_statistics_add_pdata(process_statistics *, process_data *);

extern PetscErrorCode process_statistics_add_accept(process_statistics *,
                tcpaccept_entry *);

extern PetscErrorCode process_statistics_add_connect(process_statistics *,
                 tcpconnect_entry *);

extern PetscErrorCode process_statistics_add_connlat(process_statistics *,
                 tcpconnlat_entry *);

extern PetscErrorCode process_statistics_add_life(process_statistics *,
                tcplife_entry *);

extern PetscErrorCode process_statistics_add_retrans(process_statistics *,
                 tcpretrans_entry *);

extern PetscErrorCode process_statistics_get_pid_data(process_statistics *,
                  PetscInt,
                  process_data *);


typedef struct {
  FILE *file;
  long offset;
} file_wrapper;


extern long get_file_end_offset(file_wrapper *);

/* returns 0 if no new data; if the return value is non-zero, it is the difference between the new and old end-of-file. However, the file is moved back to the old end, so you can read the new data. */
extern long has_new_data(file_wrapper *);



#endif
```