Flow Design Document

PVFS Development Team

July 2002

1 TODO

- point to some other document for explanation of concepts common to all pvfs2 I/O interfaces (contexts, max idle time, test semantics, etc.)

2 Concepts and Motivation

Flows are a high level model for how PVFS2 system components will perform I/O. It is designed to abstractly but efficiently move data from source to destination, where source and destination may be defined as storage devices, network devices, or memory regions.

Features include:

- *Combining I/O mechanisms* The flow interface combines network I/O and disk I/O into a single framework with a scheduler that takes both into account.
- *Multiple protocols* Actual I/O is carried out underneath the flow interface by *flow protocols*. We may implement several different protocols (using different I/O or buffering techniques, for example) which can be switched at runtime.
- *Simple interface* The application interface to this system will be as high level and simple as possible. Device selection, scheduling, buffer management, and request pattern processing will be transparent to the flow user.
- *Datatypes* Flows allow the user to specify both memory and file datatypes (similar to those used in MPI) , and will handle breaking down these datatypes into a format that can be used by lower level I/O interfaces.

3 Flows

3.1 Overview

A flow describes a movement of data. The data always moves from a single source to a single destination. There may be (and almost always will be) multiple flows in progress at the same time for different locations- in particular for clients that are talking simultaneously to several servers, or servers that are handling simultaneous I/O requests.

At the highest level abstraction, it is important that a flow describes a movement of data in terms of "what to do" rather than "how to do it". For example, when a user sets up a flow, it may indicate that the first 100 bytes of a file on a local disk should be sent to a particular host on the network. It will not specify what protocols to use, how to buffer the data, or how to schedule the I/O. All of this will be handled underneath the flow interface. The user just requests that a high level I/O task be performed and then checks for completion until it is done.

Note that the "user" in the above example is most likely a system interface or server implementer in pvfs2. End users will be unaware of this API.

A single flow created on a server will match exactly one flow on a client. For example, if a single client performs a PVFS2 read, the server will create a storage to network flow, and the client will create a network to memory flow. If a client communicates with N servers to complete an I/O operation, then it will issue N flows simultaneously.

Flows will not be used for exchanging request protocol messages between the client and server (requests or acknowledgements). They will only be used for data transfer. It is assumed that request messages will be used for handshaking before or after the flow as needed.

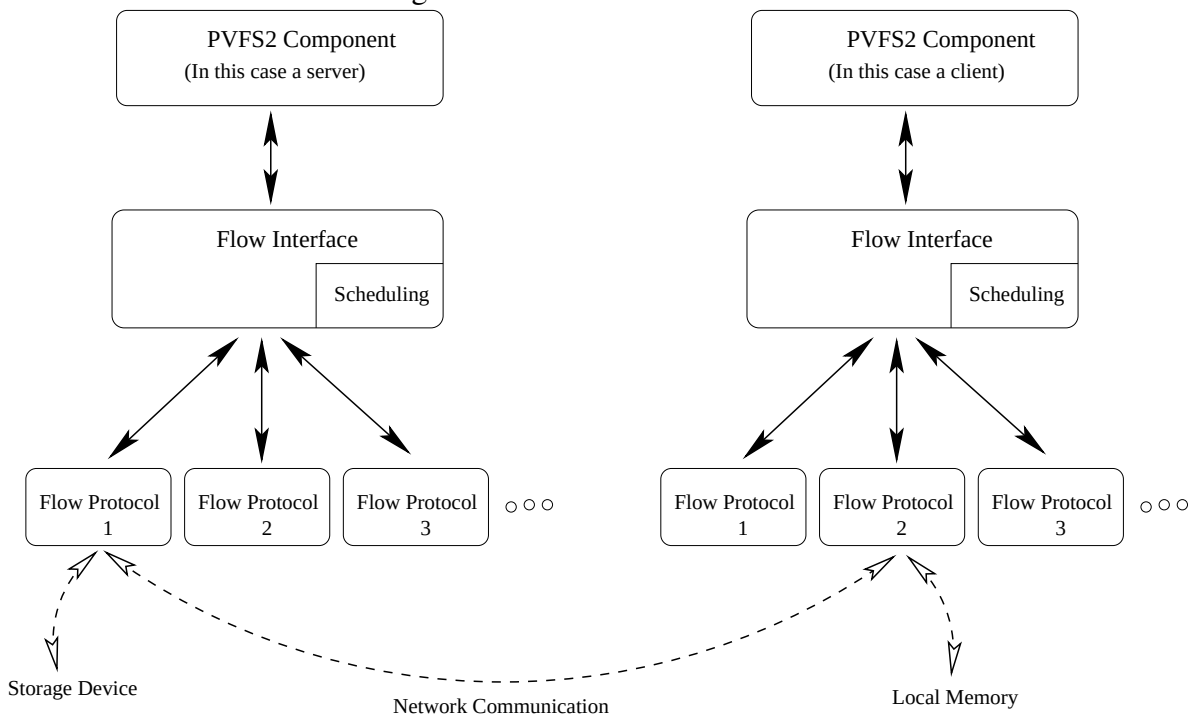
3.2 Architecture

There are two major parts of the flow architecture, as seen in figure 1. The first is the *flow interface*. Applications (ie PVFS components) interact with this interface. It provides a consistent API regardless of what protocols are in use, what scheduling is being performed, etc.

The second major component of the architecture is the *flow protocol*. There may be many flow protocols active within one flow interface. Each flow protocol implements communication between a different pair of data endpoint types. For example, one flow protocol may link TCP/IP to asynchronous unix I/O, while another may link VIA to memory regions. For two seperate hosts to communicate, they must share compatible flow protocols (as indicated by the dotted line at the bottom of figure 1).

Flow protocols all adhere to a strict interface and must provide the same expected functionality (which will be described later). Flow protocols take care of details such as buffering and flow control if necessary.

Figure 1: Basic flow architecture



3.3 Describing flows

Individual flows are represented using structures called *flow descriptors*. The source and destination of a given flow are represented by structures called *endpoints*. A flow descriptor may serve many roles. First of all, when created by a flow interface user, it describes an I/O task that needs to be performed. Once it is submitted to the flow interface, it may keep up with state or progress information. When the descriptor is finally completed and returned to the user, it will indicate the status of the completed flow, whether successful or in error.

Flow endpoints describe the memory, storage, or network locations for the movement of data. All flow descriptors must have both a source and a destination endpoint.

3.4 Usage assumptions

It is assumed that all flows in PVFS2 will be *preceded* by a PVFS2 request protocol exchange between the client and server. In a file system read case, the client will send a read request, and the server will send an acknowledgement (which among other things indicates how much data is available to be read). In a file system write case, the client will send a write request, and the server will send an acknowledgement that indicates when it is safe to begin the flow to send data to the server. Once the flow is completed, a

trailing acknowledgment alerts the client that the server has completed the write operation.

The request protocol will transmit information such as file size and distribution parameters that may be needed to coordinate remote flows.

4 Data structures

4.1 Flow descriptor

Flow descriptors are created by the flow interface user. At this time, the caller may edit these fields directly. Once the flow has been posted for service, however, the caller may only interact with the descriptor through functions defined in the flow interface. It is not safe to directly edit a flow descriptor while it is in progress.

Once a flow is complete, it is again safe to examine fields within the descriptor (for example, to determine the status of the completed flow).

Note that there is an endpoint specific to each type supported by the flow interface (currently memory, BMI (network), and Trove (storage)).

The following fields may be set by the caller prior to posting:

- src: source endpoint (BMI, memory, or Trove addressing information)
- dest: destination endpoint (BMI, memory, or Trove addressing information)
- tag: tag used to match up flows with particular operation sequences
- user_ptr: void* pointer reserved for use by the caller (may associate a flow with some higher level state structure, for example)
- type: specifies what kind of flow protocol to use for this flow
- file_req: file datatype (similar to MPI datatype)
- file_req_offset: offset into the file datatype
- mem_req: memory datatype (similar to MPI datatype) (optional)
- aggregate_size: total amount of data the flow should transfer (optional)
- file_data: struct containing state information about the file to access, used by the distribution subsystem

Special notes: Both the mem_req and the aggregate_size fields are optional. However, at least one of them *must* be set. Otherwise the flow has no way to calculate how much data must be transferred.

The following fields may be read by the caller after completion of a flow:

- state: final state of flow (see enumerated values in flow.h)
- error_code: error code (nonzero if state indicates an error)
- total_transfered: amount of data moved by the flow

The following fields are reserved for use within the flow code:

- context_id: specifies which flow level context the descriptor belongs to
- flowproto_id: internal identifier for the flowprotocol used
- priority: priority level of flow (unused as of yet)
- sched_queue_link: for use by internal scheduler
- flow_protocol_data: void* reserved for flow protocol use
- file_req_state: current state of file datatype processing
- mem_req_state: current state of memory datatype processing
- result: result of each datatype processing iteration

5 Flow interface

The flow interface is the set of functions that the flow user is allowed to interact with. These functions allow you to do such things as create flows, post them for service, and check for completion.

- *PINT_flow_initialize()*: performs initial setup of flow interface - must be called once before any other flow interface functions
- *PINT_flow_finalize()*: shuts down the flow interface
- *PINT_flow_alloc()*: creates a new flow descriptor
- *PINT_flow_free()*: frees up a flow descriptor that is no longer needed
- *PINT_flow_reset()*: resets a previously used flow descriptor to its initial state and values.

- *PINT_flow_set_priority()*: sets the priority of a particular flow descriptor. May be called even when a flow is in service.
- *PINT_flow_get_priority()*: reads the priority of a particular flow descriptor. May be called even when a flow is in service.
- *PINT_flow_post()*: submits a flow descriptor for service
- *PINT_flow_setinfo()*: used to set optional interface parameters.
- *PINT_flow_getinfo()*: used to read optional interface parameters.

Three functions are provided to test for completion of posted flows:

- *PINT_flow_test()*: tests for completion of a single flow
- *PINT_flow_testsome()*: tests for completion of any flows from a specified set of flows
- *PINT_flow_testcontext()*: tests for completion of any flows that are in service in the interface

6 Flow protocol interface

The flow protocols are modular components capable of moving data between particular types of endpoints. (See section 3.2 for an overview). Any flow protocol implementation must conform to a predefined flow protocol interface in order to interoperate with the flow system.

- *flowproto_initialize()*: Initializes the flow protocol (called exactly once before posting any flows)
- *flowproto_finalize()*: shuts down the flow protocol (forceful terminating any pending flows)
- *flowproto_post()*: posts a flow descriptor
- *flowproto_find_serviceable()*: returns an array of active flows from the flow protocol that are either completed or in need of service
- *flowproto_service()*: performs work on a single flow descriptor that is ready for service (as indicated by a flowproto_find_serviceable() function)
- *flowproto_getinfo()*: reads optional parameters from the protocol
- *flowproto_setinfo()*: sets optional protocol parameters

The following section describing the interaction between the flow component and the flow protocols may be helpful in clarifying how the above functions will be used.

7 Interaction between flow component and flow protocols

The flow code that resides above the flow protocols serves two primary functions: multiplexing between the various flow protocols, and scheduling work.

The multiplexing is handled by simply tracking all active flow protocols and directing flow descriptors to the appropriate one.

The scheduling functionality is the more complicated of the two responsibilities of the flow code. This responsibility leads to the design of the flow protocol interface and the states of the flow descriptors. In order to understand these states, it is important to understand that flow protocols typically operate with a certain granularity that is defined by the flow protocol implementation. For example, a flow protocol may transfer data 128 KB of data at a time. A simple implementation of a memory to network flow may post a network send of 128 KB, wait for it to complete, then post the next send of 128 KB, and so on. Each of these iterations is driven by the top level flow component. In other words, the flow protocol is not autonomous. Rather than work continuously once it receives a flow descriptor, it only performs one iteration of work at a time, and then waits for the flow component to tell it to continue. This provides the flow interface with an opportunity to schedule flows and choose which ones to service at each iteration.

When a flow descriptor is waiting for the flow component to allow it to continue, then it is "ready for service". The flow component may then call the flowproto_service() function to allow it to continue. In the above example, this would cause the flow protocol to post another network send.

In order to discover which flow descriptors are "ready for service" (and therefore must be scheduled), it calls flowproto_find_serviceable() for each active flow protocol. Thus, the service loop of the flow component looks something like this:

1. call flowproto_find_serviceable() for each active flow protocol to generate a list of flows to service
2. run scheduling algorithm to build list of scheduled flows
3. call flowproto_service() for each scheduled flow (in order)
4. if a flow descriptor reaches the completed or error state (at any time), then move it to a list of completed flow descriptors to be returned to the caller

The scheduling filter (at the time of this writing) does nothing but service all flows in order. More advanced schedulers will be added later.

7.1 Example flow protocol (implementation)

The default flow protocol is called "flowproto_bmi_trove" and is capable of handling the following end-point combinations:

- BMI to memory
- memory to BMI
- BMI to Trove
- Trove to BMI

The following summarizes what the principle flow protocol interface functions do in this protocol:

- flowproto_post(): allocates any intermediate buffers that may be needed, begins datatype processing
- flowproto_service(): posts the next necessary BMI and Trove operations
- flowproto_find_serviceable(): tests for completion of pending BMI and trove operations, drives the state of any flow descriptors affected by completion, and returns flow descriptors ready for service to caller

The flow protocol performs double buffering to keep both the Trove and BMI interfaces as busy as possible when transferring between the two.

The flow protocol does not have an internal thread. However, if it detects that the job interface is using threads (through the __PVFS2_JOB_THREADED__ define), then it will use the job interface's thread manager to push on BMI and Trove operations. The find_serviceable() function then just checks for completion notifications from the thread callback functions, rather than testing the BMI or Trove interfaces directly.

Trove support is compiled out if the __PVFS2_TROVE_SUPPORT__ define is not detected. This is mainly done in client libraries which do not need to use Trove in order to reduce library dependencies.

8 Implementation note: avoiding flows

The flow interface will introduce overhead for small operations that would not otherwise be present. It may therefore be helpful to eventually introduce an optimization to avoid the use of flows for small read or write operations.

```
text of an email discussion on this topic (> part by Phil, non >
part by Rob):

> Yeah, we need to get these ideas documented somewhere.  There may actually
```

> be a couple of eager modes. By default, BMI only allows unexpected
> messages < 16K or so. That places a cap on the eager write size,
> unless we had a second eager mode that consists of a) send write request
> b) send write data c) receive ack...

Yes. These two modes are usually differentiated by the terms "short" and "eager", where the "short" one puts the data actually into the same packet/message (depending on the network layer at which we are working).

> Of course all of this would need to be tunable so that we can see what
> works well. Maybe rules like:
>
> contig writes < 15K : simple eager write
> 15K < contig writes < 64K : two part eager write
> writes > 64K && noncontig writes : flow
>
> contig reads < 64K : eager read
> contig reads > 64K && noncontig reads : flow

Yeah, something like that.