tgt: framework for storage target drivers

FUJITA Tomonori
NTT Cyber Solutions Laboratories
tomof@acm.org

Mike Christie
Red Hat, Inc.
michaelc@cs.wisc.edu

Abstract

In order to provide block I/O services, Linux users have had to modify kernel code by hand, use binary kernel modules, or purchase specialized hardware. With the mainline kernel now having SCSI Parallel Interface (SPI), Fibre Channel (FC), iSCSI, and SCSI RDMA (SRP) initiator support, Linux target framework (tgt) aims to fill the gap in storage functionality by consolidating several target driver implementations and providing a SCSI protocol independent API that will simplify target driver creation and maintenance.

Tgt's key goal and its primary hurdle has been implementing a great portion of tgt in user space, while continuing to provide performance comparable to a target driver implemented entirely in the kernel. By pushing the SCSI state machine, I/O execution, and the management components of the framework outside of the kernel, it enjoys debugging, maintenance and mainline inclusion benefits. However, it has created new challenges. Both traditional kernel target implementations and tgt have had to transform Block Layer and SCSI Layer designs, which assume requests will be initiated from the top of the storage stack (the request queue's `make_request_fn()`) to an architecture that can efficiently handle asynchronous requests initiated by the the end of the stack (the low level drivers interrupt handler), but tgt also must efficiently communicate and synchronize with the user-space daemon that implements the SCSI target state machine and performs I/O.

1 Introduction

The SCSI protocol was originally designed to use a parallel bus interface and used to be tied closely to it. With the increasing demands of storage capacity and accessibility, it became obvious that Direct Attached Storage (DAS), the classic storage architecture, in which a host and storage devices are directly connected by system buses and parallel cable, cannot meet today's industry scalability and manageability requirements. This lead to the invention of Storage Area Network (SAN) technology, which enables hosts and storage devices to be connected via high-speed interconnection technologies such as Fibre Channel, Gigabit Ethernet, Infiniband, etc.

To enable SAN technology, the SCSI-3 architecture, as can be seen in Figure 1, brought an important change to the division of the standard into interface, protocol, device model, and command set. This allows device models and command sets with various transports (physical interfaces), such as Fibre Channel, Ethernet, and Infiniband. The device type specific command set, the primary command set, and transport are independent of each other.

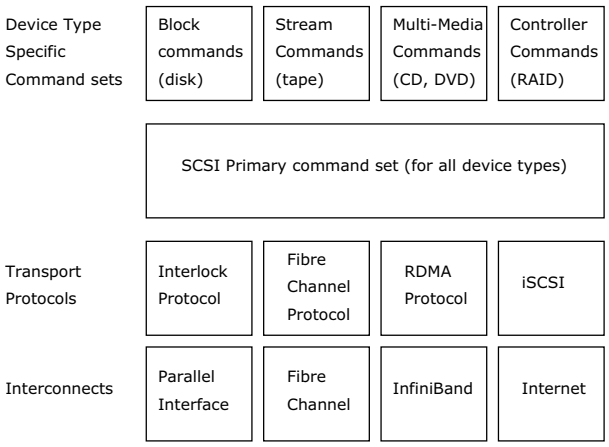

Figure 1: SCSI-3 architecture

1.1 What is a Target

SCSI uses a client-server model (Figure 2). Requests are initiated by a client, which in SCSI terminology is called a Initiator Device, and are processed by a server, which in SCSI terminology is known as a Target Device. Each target contains one or more logical units and provides services performed by device servers and task management functions performed by task managers. A logical unit is a object that implements one or more device functional models described in the SCSI command standards and processes commands (eq., reading from or writing to the media) [5].

Currently, the Linux kernel has support for several types of initiators including ones that use FC, TCP/IP, RDMA, or SPI for their transport protocol. There is however, no mainline target support.

2 Overview of Target Drivers

2.1 Target Driver

Generally in the past, a target driver is responsible for the following tasks:

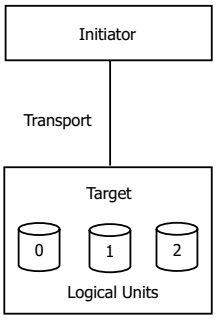

Figure 2: SCSI target and initiator

1. Handling its interconnect hardware interface and transport protocol.
2. Processing the primary and device specific command sets.
3. Accessing local devices (attached to the server directly) when necessary.

Since hardware interfaces are unique, the kernel needs a specific target driver for every hardware interface. However, the rest of the tasks are independent of hardware interfaces and transport protocols.

The duplication of code between task two and three lead to the necessity for a target framework that provides a API set useful for every target driver. In tgt, target drivers simply take SCSI commands from transport protocol packets, hand them over to the framework, and send back the responses to the clients via transport protocol packets. Figure 3 shows a simplified view of how hardware interfaces and transport protocols interact in tgt. It is more complicated than the above explanation of the ideal model due to some exceptions described below.

Tgt is integrated with Linux's SCSI Mid Layer (SCSI-ML), so it supports two hardware interface models:

Hardware A Host Bus Adapter (HBA) handles the major part of transport protocol

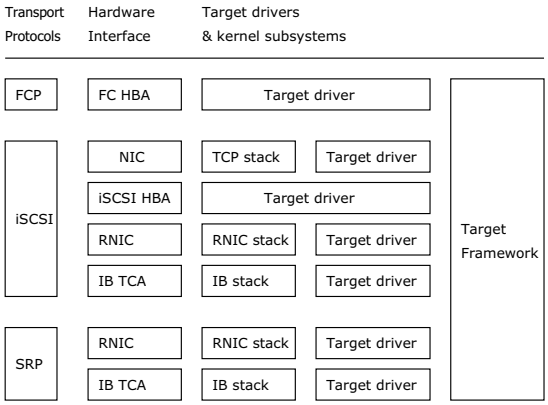

Figure 3: transport protocols and hardware interfaces

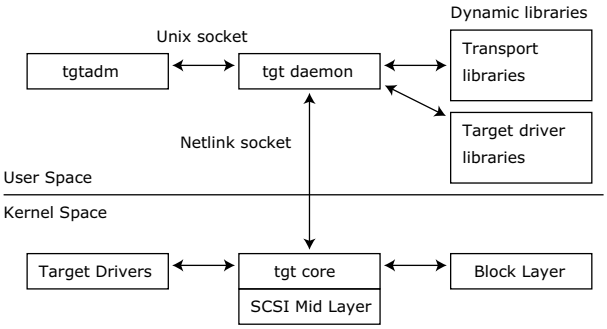

Figure 4: tgt components

processing and the target driver implements the functionality to communicate between the HBA and tgt. Tgt needs a specific target driver for each type of HBA. FCP and SPI drivers follow this model. Drivers for other transports like iSCSI or SRP or for interconnects like iSER follow this model when there is specialized hardware to offload protocol or interconnect processing.

Software For transports like iSCSI and SRP or interconnects like iSER, a target driver can implement the transport protocol processing in a kernel module and access low level hardware through another subsystem such as the networking or infiniband stack. This allows a single target driver to work with various hardware interfaces.

3 Target Framework (tgt)

Our key design philosophy is implementing a significant portion of tgt in user space while maintaining performance comparable to a target driver implemented in kernel space. This conforms to the current trend of pushing code that can be implemented in user space out of the kernel [6] and enables developers to use rich user space libraries and development tools such as gdb.

As can be seen in Figure 4, the tgt architecture has two kernel components: the target driver and tgt core. The target driver's primary responsibilities are to manage the transport connections with initiator devices and pass commands and task management function requests between its hardware or interconnect subsystem and tgt core. tgt core is a simple connector between target drivers and the user space daemon (tgtd) that enables the driver to send tgtd a vector of commands or task management function requests through a netlink interface.

Tgt core was integrated into scsi-ml with minor modifications to the `scsi_host_template` and various scsi helpers functions for allocating scsi commands. This allows tgt to rely on scsi-ml and the Block Layer for tricky issues such as hot-plugging, command buffer mapping, scatter gather list creation, and transport class integration. Note that tgt does not change the current scsi-ml API set, so normally the only modifications are required to the initiator low level driver's (LLD) interrupt handler to process target specific requests and to the transport classes so that they are able to present target specific attributes.

All SCSI protocol processing is performed in

user space, so as can be seen by Figure 4 the bulk of the tgt is implemented in: tgtadm, tgtd, transport libraries and driver libraries. tgtadm is a simple management tool. A transport library is equivalent to a kernel transport class where functionality common to a set of drivers using the same transport can be placed. Driver libraries, are dynamically linked target driver specific libraries that can be used to implement functionality such as special setup and tear down operations. And, tgtd is the SCSI state machine that executes commands and task management requests.

The clear concern over the user space SCSI protocol processing is degraded performance[1]. We explain some techniques to overcome this problem as we discuss in more detail the tgt components.

3.1 API for Target Drivers

The target drivers interact with tgt core through a new tgt API set, and the existing mid-layer API set and data structures. For the most part, target drivers work in a very similar manner as the existing initiator drivers. In many cases the initiator only needs to implement the new target callbacks on the `scsi_host_template`: `transfer_response()`, `transfer_data()`, and `tsk_mgmt_response()`, to enable a target mode in its hardware. We examine the details of the new callbacks later in this section.

3.1.1 Kernel Setup

The first step in registering a target driver with scsi-ml and tgt core is to create a scsi host adapter instance. This is accomplished by calling the same functions that are used for the initiator: `scsi_host_alloc()` and `scsi_add_host()`. If a HBA will be running in both target and initiator mode then only a single call to each of those functions is necessary for each HBA. The final step in setting up a target driver is to allocate a `uspace_req_q` for each scsi host that will be running in target mode. A `uspace_req_q` is used by tgt core to send requests to users-space. It can be allocated and initialized by calling `scsi_tgt_alloc_queue()`.

3.1.2 Processing SCSI Commands in the Target Driver

The target driver needs to allocate the `scsi_cmnd` data structure for a SCSI command received from a client via `scsi_host_get_command()`. This corresponds to scsi-ml's `scsi_get_command()` usage for allocating a `scsi_cmnd` for each request coming from the Block Layer or scsi-ml Upper Layer Driver (ULD). While the former allocates the `scsi_cmnd` and the `request` data structures, the latter allocates only the `scsi_cmnd` data structure.

The target driver sets up and passes the `scsi_cmnd` data structure to tgt core via `scsi_tgt_queue_command()`. The following information is passed to tgt core from the target driver:

SCSI command buffer to contain SCSI command.

lun buffer buffer to represent logical unit number.

tag unique value to identify this SCSI command.

[1] In the early days, tgt performed performance sensitive SCSI commands in kernel space (eq. read/write from/to storage devices). However, it turned out that the current design was able to achieve comparable performance.

task attribute task attribute for ordering.

buffer length number of data bytes to transfer.

On completion of executing a SCSI command, tgt core invokes `transfer_response()`, which is specified in the `scsi_host_template` data structure.

`transfer_data()` is invoked prior to `transfer_response()` if a SCSI command involves data transfer. Like scsi-ml, a scatter gather list of pages at the `request_buffer` member in the `scsi_cmnd` data structure is used to specify data to transfer. Also like scsi-ml, tgt core utilizes Block Layer and scsi-ml helpers to create scatter gather lists within the `scsi_host_template` limits such as `max_sectors`, `dma_boundary`, `sg_tablesize`, and `use_clustering`.

If the SCSI command involves a target-to-initiator data transfer, a target driver transfers data pointed out by the scatter gather list to the client, and then invokes the function pointer passed as a argument of `transfer_data()` to notify tgt core of the completion of the operation.

If the SCSI command involves a initiator-to-target data transfer, the target driver copies (through a DMA operation or memcpy) data to the scatter gather list (the LLD or transport class requests the client to send data to write before the actual transfer if necessary), and then invokes the function pointer passed as a argument of `transfer_data()` to notify tgt core of the completion of the transfer.

Depending on the transfer size and hardware or transport limitations, tgt core may have to call `transfer_data()` multiple times to transmit the entire payload. To accomplish this, tgt is not able to easily reuse the existing Block Layer and SCSI API. This is due to tgt core executing from interrupt context, and because the scatter list APIs tgt utlizes were not intended for requests starting at end of the storage stack. To work around the Block Layer scatter gather list allocation function assumption that a request will normally be completed in one scatter list, tgt required two modifications or workarounds. The first and easiest, was the addition of an `offset` field to the `scsi_cmnd` to track where the LLD is currently at in the transfer. The more difficult change, and probably more of a hack, was for tgt core to maintain two lists of BIOs for each request. One list contains BIOs that have not been mapped to scatter lists and the second list contains BIOs that have been mapped into scatter gather lists, completed, and need to be unmapped from process context when the command is completed.

3.1.3 Task Management Function

A target driver can send task management function (TMF) requests to tgt core via `scsi_tgt_tsk_mgmt_request()`.

The first argument is the TMF type. Currently, the supported TMF types are `ABORT_TASK`, `ABORT_TASK_SET`, and `LOGICAL_UNIT_RESET`.

The second argument is the tag value to identify a command to abort. This corresponds to the tag argument of `scsi_tgt_queue_command()` and used only with `ABORT_TASK`.

The third argument is the lun buffer to identify a logical unit against which the TMF request is performed. This is used with TMF requests except for `ABORT_TASK`.

The last argument is a pointer to enable target drivers to identify this TMF request on completion of it.

tgt core invokes `eh_abort_handler()` per aborted command to allow the target driver to clean up any resources that it may have internally allocated for the command. Unlike when it is called by scsi-ml's error handler, the host is not guaranteed to be quiesced and may have initiator and target commands running.

Subsequently to `eh_abort_handler()`, `tsk_mgmt_response()` is invoked. The pointer to identify the completed TMF request is passed as the argument.

3.2 tgt core

tgt core conveys SCSI commands, TMF requests, and these results between target drivers and the user space daemon, tgtd, through a netlink interface, which enables a user space process to read and write a stream of data via the socket API. tgt core encapsulates the requests into netlink packets and sends them to user space to be executed. Then it receives netlink packets from user space, extracts the results of the operation, and performs auxiliary tasks in compliance with the results. Figure 5 shows the packet format for SCSI commands.

```
struct {
    int host_no;
    uint32_t cid;
    uint32_t data_len;
    uint8_t scb[16];
    uint8_t lun[8];
    int attribute;
    uint64_t tag;
} cmd_req;
```

Figure 5: Netlink packet for SCSI commands

Since moving large amounts of data via netlink leads to a performance drop because of the memory copies, for the command's data buffer tgt uses the memory mapped I/O technique utilized by Linux SCSI generic (SG) device driver [4], which moves an address that the `mmap()` system call returns instead of lots of data.

When tgt core receives the address from user space, it increments the reference count on the pages of the mapped region and sets up the the scatter gather list in `scsi_cmnd` data structure. tgt core relies on the standard kernel API, `bio_map_user()`, `scsi_alloc_sgtable()`, and `blk_rq_map_sg()` for these chores. Similarly, `bio_unmap_user()` and `scsi_free_sgtable()` decrements the reference and cleans up the the scatter gather list. The former also marks the pages as dirty in case of initiator-to-target data transfer (`WRITE_*` command).

3.3 User Space Daemon (tgtd)

The user space daemon, tgtd, is the heart of tgt. It contains the SCSI state machine, executes requests and provides a consistent API for management via Unix domain sockets. It communicates with the target drivers through tgt core's netlink interface.

tgtd currently uses a single process model. This enables us to avoid tricky race conditions. Imagine that a SCSI command is sent to a particular device and a management request to remove the device comes at the same time. However, this means that tgtd always needs to work in an asynchronous manner.

The tgtd code is independent of transport protocols and target drivers. The transport-protocol dependent and target-driver dependent features, such as showing parameters, are implemented in dynamic libraries: transport-protocol libraries and target-driver libraries.

There are two instances that the administrators must understand: target and device. A target

instance works as a SCSI target device server. Every working scsi host adapter that implements a target driver is bound to a particular target instance. Multiple scsi host adapter instances can be bound to a single target instance. A device instance corresponds to a SCSI logical unit. A target instance can have multiple device instances.

3.3.1 SCSI Command Processing in tgtd

In previous sections, the process by which a command is moved between the kernel and user space and how it is transferred between the target and initiator ports has been detailed. Now, the final piece of the process, where tgtd performs command execution, is described.

1. tgtd receives a netlink packet containing a SCSI command and finds the the target instance (the device instance is looked up if necessary) to which the command should be routed. As shown in Figure 5, the packet contains the host bus adapter ID and the logical unit buffer.
2. tgtd processes the task attribute to know when to execute the command (immediately or delay).
3. When the command is scheduled, tgtd executes it and sends the result to tgt core.
4. tgtd is notified via tgt core's netlink interface that the target driver has completed any needed data transfer and has successfully sent the response. tgtd is then able to free resources that it had allocated for the command.

In case of non-I/O commands, involving target-to-initiator data transfer, tgtd allocates buffer via `valloc()`, builds the response in it, and sends the address of the buffer to tgt core. The buffer is freed on completion of the command.

In case of I/O commands, tgtd maps the requested length staring at the offset from the device's file, and sends the address to the tgt core. On completion of the command, tgt calls the `munmap` system call.

To improve performance, if tgtd can map the whole device file (typically, it is possible with 64-bit architectures), tgtd does not call the `mmap` or `munmap` system calls per command. Instead, it maps the whole device file when the device instance is added to a target instance.

3.3.2 Task Management Function

When tgtd receives task management function requests to abort SCSI commands, it searches the commands, sends an abort request per the found commands, and then sends the TMF completion notification to tgt core.

Once tgt core marks pages as a dirty, it is impossible to stop them being committed to disk. Thus, tgtd does not try to abort a command if it is waiting for the completion. If tgtd receives a request to abort such command, it waits for the completion of the command and then sends the TMF completion notification indicating that the command is not found.

3.4 Configuration

The currently supported management operations are: creation and deletion of target and device instances and binding a host adapter instance to a target instance. All objects are independent of transport protocols. Transport-protocol dependent management requests (such as showing parameters) are performed by using the corresponding transport-protocol library.

The command-line management tool, *tgtadm*, is distributed together with tgt for ease of use, though tgtd provides a management API via Unix domain sockets so that administrators or vendors can implement their own management tools.

4 Status and the Future

Today, tgt implements only what is necessary to be able to benchmark it against kernel driver implementations and provide basic functionality. This has been due to the code being destabilized several times as a result of code review comments that have forced most of the code to be pushed to user space. This means that there is a long list of features to be implemented and ported from previous versions of tgt.

4.1 Target Driver Support

At the time of the writing of this paper, there is only one working target driver, ibmvstgt, which is a SRP target driver, though it works only for virtualization environments on IBM pSeries [2]. The virtual machines communicate via RDMA. One virtual machine (called the Virtual I/O server) works as a SRP server that provides I/O services for the rest of virtual machines. ibmvstgt is based on IBM standalone (not a framework) target driver [3], ibmvscsis. By converting ibmvscsis to tgt more than 2,000 lines were removed from the original driver.

Currently, there is a Qlogic FC, qla2xxx based, target driver being converted from kernel based target framework, SCST [9], to tgt. And, a Emulex FC, lpfc based, target driver that utilized a GPL FC target framework is being worked on. Both of these require FC transport class Remmote Port (rport) changes that will allow the FC class and tgt core to perform transport level recovery for the target LLDs.

The iSCSI code in mainline has also begun to be modified to support target mode. With the introduction of libiscsi a target could be implemented by creating a new iscsi transport module or by modifying the iscsi tcp module.

4.2 Kernel and User Space Communication

Netlink usage leads to memory allocations and memory copies for every netlink packet. It also suffers from frequent system calls. tgt avoids a significant performance drop by using the memory mapped I/O technique for the command data buffer, but there is still room for improvement. By removing the copy of the command and its status and sending a vector of commands or status we can reduce the memory copies and kernel user space trips.

Previous versions of tgt had used a mmapped packet socket (`AF_PACKET`), to send messages to user space. This removes netlink from the command execution initiation and proved to be a performance gain, but difficulties in the mmapped packet socket interface usage have prevented tgt from currently using it. Another option that provides high speed data transfers is the relayfs file system [10]. Unfortunately, both are unidirectional, and only communication from the kernel to user space is improved.

Another common technique for reducing system call usage is to send multiple requests in one invocation. During testing of tgt, this was attempted and showed promise, but was difficult to tune. Investigation will be resumed when more drivers are stabilized and can be benchmarked.

4.3 Virtualization

The logical unit that executes requests from a remote initiator is not limited to being backed by a local device that provides a SCSI interface. The object being used by tgt for the logical unit's device instance could be a IDE, SATA, SCSI, Device Mapper (DM), or Multiple Device (MD) device or a file. To provide this flexibility, previous versions of tgt provided *virtualized devices* for the clients regardless of the attached local object. tgtd had two types of virtualization:

device virtualization With device virtualization, a `device_type_handler` (DTH) emulates commands that cannot be executed directly by its device type. For example, a MD or DM device has no notion of a SCSI INQUIRY. In this case the DTH has the generic SCSI protocol emulation library execute the INQUIRY.

I/O virtualization With I/O virtualization a `io_type_handler` (IOTH) enables tgt to access regular and block device files using mmap or use specialized interfaces such as SG IO.

Many types and combinations of device and I/O virtualization are possible. A Virtual Tape Library (VTL) could provide a tape library using disk drives, or by using the virtual CD device and file I/O virtualization a tgt machine could provide cdrom devices for the client with ISO image files.

Another interesting virtualization mechanism is *passthrough*, directly passing SCSI commands to SCSI devices. This provides a feature, called storage bridge, to bind different SAN protocols. For example, suppose that you are already in a working FC environment, where there are is FC storage server and clients with a FC HBA. If you need to add new clients that need to access the FC storage server, however you cannot afford to buy new FC HBAs, an iSCSI-FC bridge can connect the existing FC network with a new iSCSI network.

Currently, tgt supports disk device virtualization and file I/O virtualization. The file I/O virtualization simply opens a file and accesses its data via the `mmap` system call.

5 Related Work

SCST is the first GPLed attempt to implement a complete framework for target drivers. There are several major differences between tgt and SCST: SCST is mature and contains many features, all the SCST components reside in kernel space, and SCST duplicates functionality found in scsi-ml and the Block Layer instead of exploiting and modifying those subsystems.

Besides ibmvscsis, there have been several standalone target drivers. iSCSI Enterprise Target software (IET) [1] is a popular iSCSI target driver for Ethernet adapters, which tgt has used as a base for istgt[2]. Other software iSCSI targets include the UNH and Intel implementations [8, 7] and there are several iSCSI and FC drivers for specific hardware like Qlogic and Chelsio.

6 Conclusion

This paper describes tgt, a new framework that adds storage target driver support to the SCSI

[2]The first author has maintained IET. The failure to push it into the mainline kernel is one of the reasons why tgt was born.

subsystem. What differentiates tgt from other target frameworks and standalone drivers is its attempt to push the SCSI state model and I/O execution to user space.

By using the Block and SCSI layer, tgt has been able to quickly implement a solution that bypasses performance problems that result from executing memory copies to and from the kernel. However, the Block and SCSI Layers, were not designed to handle large asynchronous requests originating from the LLDs interrupt handlers. Since the Block Layer SG IO and SCSI Upper Layer Drivers like SG, share a common technique, code, and problems, we hope we will be able to find a final solution that will benefit tgt core and the rest of the kernel.

Tgt has undergone several rewrites as a result of code reviews, but is now reaching a point where hardware interface vendors and part time developers are collaborating to solidify tgt core and tgtd, implement new target drivers, make modifications to other kernel subsystems to support tgt, and implement new features.

The source code is available from `http://stgt.berlios.de/`

References

[1] iSCSI Enterprise Target software, 2004. `http://iscsitarget.sourceforge.net/`.

[2] Dave Boutcher and Dave Engebretsen. Linux Virtualization on IBM POWER5 Systems. In *Ottawa Linux Symposium*, pages 113–120, July 2004.

[3] Dave Boutcher. SCSI target for IBM Power5 LPAR, 2005. `http://patchwork.ozlabs.org/linuxppc64/patch?id=2285`.

[4] Douglas Gilbert. *The Linux SCSI Generic (sg) Driver*, 1999. `http://sg.torque.net/sg/`.

[5] T10 Technical Editor. SCSI architecture model-3, 2004. `http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf`.

[6] Edward Goggin, Alasdair Kergon, Christophe Varoqui, and David Olien. Linux Multipathing. In *Ottawa Linux Symposium*, pages 147–167, July 2005.

[7] Intel iSCSI Reference Implementation, 2001. `http://sourceforge.net/projects/intel-iscsi/`.

[8] UNH-iSCSI initiator and target, 2003. `http://unh-iscsi.sourceforge.net/`.

[9] Vladislav Bolkhovitin. Generic SCSI Target Middle Level for Linux, 2003. `http://scst.sourceforge.net/`.

[10] Karim Yaghmour, Karim Yaghmour, Robert Wisniewski, Richard Moore, and Michel Dagenais. relayfs: An efficient unified approach for trasmitting data from kernel to user space. In *Ottawa Linux Symposium*, pages 519–531, July 2003.