Communicating with VICE

08 November 83 03:38

M. Satyanarayanan

Information Technology Center
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Table of Contents

List of Figures

```
SSSSSSS                    Satya,M

Printer Ruby

Spruce version 12.0 -- spooler version 12.0

File: vproto2.press on the CS-ZOG

Creation date: Tue Nov  8 03:41:31 1983

Printing date:  8-Nov-83  3:38:18 EST

For: Satya,M

Problems encountered:
28 total sheets = 22 pages, 1 copy.
Not impl.: brightness, hue, saturation, show-object, show-d[illegible]paq
Not impl.: brightness, hue, saturation, show-object, show-d[illegible]paq
Not impl.: brightness, hue, saturation, show-object, show-d[illegible]paq
Not impl.: brightness, hue, saturation, show-object, show-d[illegible]pac
Not impl.: brightness, hue, saturation, show-object, show-d[illegible]pac
Not impl.: brightness, hue, saturation, show-object, show-d[illegible]pac
Not impl.: brightness, hue, saturation, show-object, show-d[illegible]pac
Not impl.: brightness, hue, saturation, show-object, show-d[illegible]pac
```

1 Introduction

The fundamental question addressed in this document is "*How does one enable a process running on a network node to reliably and securely obtain services from a process at a remote site?*"

Any communication scheme that attempts to meet this requirement must incorporate the following capabilities:

- a mechanism to physically ***transport*** data from a process on one machine to a process on another.
- a network-wide ***naming*** scheme, allowing a requestor to uniquely identify a process or class of processes providing a service.
- a mechanism to allow a requestor and a server processes on remote machines to ***rendezvous*** prior to data transmission.
- an ***application protocol*** that allows a requestor and server to meaningfully interpret what each is saying to the other.

A variety of transport protocols are available today to meet the first of these requirements. The DARPA Internet family [3, 4, 5] and the SNA [2] family are examples of such protocols. Since these protocols are well-defined and the problems addressed by them well-understood, we shall not discuss them further in this paper.

The second and third requirements (naming and rendezvous) are addressed in Unix 4.2BSD via an abstraction called a ***Socket***. This abstraction is adopted as the basis for the communication scheme described in this paper. An abbreviated description of the socket mechanism, as pertinent to this discussion, is presented in Section 2. The issue of communicating with non-Unix network nodes is also discussed in that section.

When mature, VICE will consist of a number of independent subsystems providing a variety of services. In designing an application protocol for VICE, one has to strike a balance between generality and efficiency. The model described in Section 3 attempts to tread this thin line carefully, using the socket mechanism as its basis.

In reading this document please bear in mind that this is a skeleton: many details are omitted and a number of fine points glossed over. It is hoped, however, that the presentation here is in sufficient depth to convey the feasibility and adequacy of the proposed communication scheme.

2 The Socket Mechanism in Unix

A detailed discussion of the Unix socket mechanism is presented in [8]. The material presented here is essentially a simplified summary of that document, with minimal attention being paid to implementation issues. A discussion of the issues arising in an actual implementation may be found in [9].

2.1 Sockets

A ***Socket*** is an abstraction representing one end of a bidirectional communication between two Unix processes. These processes may be located on the same machine, or on different machines connected by a network and capable of exchanging data via a common transport protocol. A socket has a global name called its ***Socket Address*** which is unique across the entire network. A socket may also have one or more process-specific local names called ***Socket Descriptors***. The scheme is an exact analogue of the Unix file system, where file names are global but file descriptors are process-specific.

With a view to supporting a multiplicity of transport protocols, naming schemes, and service guarantees, a socket is associated with the following attributes:

- a ***Domain***, defining the format of socket addresses.
- a ***Type***, characterizing the service guarantees provided by a socket.
- a ***Protocol***, specifying the lower-level transport protocol associated with transmissions via a socket.

For the purpose of this discussion we shall assume that all sockets in the network are in the same domain. The specific format of the socket addresses is immaterial here. We also assume that a single low-level transport protocol is being used. This is a weaker assumption than the previous one, and may be relaxed without loss of generality.

Currently Unix supports two types of sockets: ***Stream Sockets*** and ***Datagram Sockets***. At any instant of time, a stream socket may be bound to at most one other stream socket in the domain. The two connected sockets adhere to client and server roles, as described in Section 2.3. Data transmissions on such a connection are error-free, flow-controlled, sequenced, and devoid of packet boundaries — the abstraction provided is that of a stream of bytes. In contrast, transmissions via a datagram socket are liable to be lost or garbled, or may arrive out of sequence at their destinations. Packet boundaries in such transmissions are preserved, since each input or output operation on a datagram socket deals with an entire packet.

2.2 Creating and Naming Sockets

A process first creates a socket descriptor and then binds this descriptor to a socket address. After all operations on the socket are complete, a process may rid itself of the socket descriptor. The socket itself will not vanish until all socket descriptors referring to it vanish.

The Unix operations corresponding to these actions are:

- *socket descriptor* = **socket** (*domain*, *type*, *protocol*)
- **bind** (*socket descriptor*, *socket address*, *address length*)
- **close** (*socket descriptor*)
- **shutdown** (*socket descriptor*, *how*)[1]

2.3 Stream Socket Conections

The stream socket connection model assumes that one of the interested parties is a ***Client*** and the other is a ***Server***. A server process is always listening on its socket for clients. The rendezvous between a client and a server results in the automatic creation of a new socket, and in the client being connected to it without any explicit actions on its part. At this point the server typically forks a copy of itself. The child server process services the client via the new socket, while the parent continues to listen for other clients on the original socket. On completion of service, the child server process self-destructs.

For the client process the only relevant operation is:

- **connect** (*socket descriptor*, *server socket address*, *address size*)

For the server process the corresponding operations are:

- **listen** (*socket descriptor*, *maximum number of waiting clients*)
- *new socket descriptor* = **accept** (*socket descriptor*, *client socket address*, *address len*)

2.4 Datagram Socket Connections

Whereas stream sockets must be connected prior to data transmission, datagram sockets may used in connected or connectionless modes. When connected, the connection information merely serves as routing information. There is no implied client/server relationship, nor is there any automatic creation of new sockets. When used connectionless, destination and source addresses have to be explicitly specified when transmitting data, and explicitly requested when receiving it.

[1]This is an alternative to **close**

2.5 I/O on Sockets

The actual transmission of data is done with Unix calls which closely resemble operations on file descriptors. In the case of datagram sockets, each such operation results in the sending or receiving of a packet. With stream sockets, such packet boundaries are invisible. The operations pertinent to stream sockets and connected datagram sockets are:

- **write** (*socket descriptor*, *buffer*, *buffer size*)
- **read** (*socket descriptor*, *buffer*, *buffer size*)
- **send** (*socket descriptor*, *buffer*, *buffer size*, *special flags*)
- **recv** (*socket descriptor*, *buffer*, *buffer size*, *special flags*)

The **send** and **receive** calls are identical to **write** and **read** except for the *special flags*. The function served by these flags is described in [8].

On unconnected datagram sockets, the relevant operations are:

- **sendto** (*socket descriptor*, *buffer*, *buffer size*, *flags*, *to address*, *address len*)
- **recvfrom** (*socket descriptor*, *buffer*, *buffer size*, *flags*, *from address*, *address len*)

2.6 Auxiliary Support

Besides these basic operations, Unix also provides a number of system calls intended for network support. These functions may be briefly summarized as functions for:

- locating a host, given its string name.
- locating a server's socket address, given its string name.
- identifying a protocol, in an environment supporting multiple transport protocols.
- identifying a network, in an internet environment.

There is also a multiple-wait function, to allow a process to wait for events on a number of selected sockets.

2.7 Non-Unix Workstations

The design presented in this paper assumes that all network nodes are capable of participating in socket operations. In order to attach a non-Unix workstation to VICE, it is at least necessary to implement software to respond to socket requests from other sites. It is also necessary to provide some means for programs on such a workstation to generate socket requests to other sites.

Should the socket interface available to programs on the workstation be identical to that in Unix? The answer to this question depends on how closely the operating system on the workstation resembles Unix. If the implementation would be simple and efficient, a full-fledged Unix socket abstraction would clearly be the right choice. Otherwise one will have to be satisfied with a socket interface to the outside world, while providing a different abstraction internally. There is probably no general solution to this problem — only a case-by-case examination will reveal the favoured approach for a particular kind of workstation.

A relevant issue here is the incorporation of new transport protocols, such as the LU6.2 SNA protocol, into this design. The view taken here, as detailed in [10], is that all such protocols should be made available to Unix programs via the socket abstraction.

Assuming network-wide socket support *a la Unix*, the next section describes the application protocol. The problem is first examined in the abstract, and its relationship to sockets is introduced in Section 3.4.

3 The VICE Communication Model

The VICE communication model describes a process-to-process application protocol that enables a workstation to communicate with, and obtain services from, a VICE subsystem on a cluster server. The model is essentially a Remote Procedure Call model, with the workstation process playing the role of caller, and the VICE subsystem the role of callee. There is no sharing of address spaces between caller and callee: input parameters are passed by value, and output parameters are passed by result. An area of particular concern in this model is the efficient transmittal of large-sized parameters, such as entire files.

A ***Connection*** defines a caller-callee pair, and comes into being when a workstation initiates a ***Connect*** sequence with a VICE subsystem. It terminates either with a ***Disconnect*** initiated by the workstation or on network partition between the two connected parties. Some of the important characteristics of a connection are:

- Many remote procedure calls may be made during the lifetime of a connection.
- There is no requirement that one procedure call complete before another is issued: responses from the callee are tagged in a manner that permits the caller to match its outstanding calls with incoming responses.
- All actions and side effects associated with a remote procedure call are completed before the corresponding response is returned.

- From the point of view of the caller and the callee, a connection appears to have the characteristics of a reliable communication medium. The underlying mechanisms may, however, be built on top of unreliable transport protocols.

3.1 Channels

A connection is associated with two bidirectional ***Communication Channels***: a ***Request Channel*** and a ***Bulk-Transfer Channel***.

The bulk-transfer channel is used for the sole purpose of transmitting large-sized parameters, generically referred to as ***Bulk-Units***. All other data associated with a connection is transmitted over the request channel. The presence of two channels is intended to accomplish the following:

- Specialized protocols may be used to efficiently transfer large-sized parameters over the bulk-transfer channel.

- The endpoints of the two channels can be distinct. For example, two high-level processes may agree to transfer a file between the two machines they are running on. The actual transfer of the file may occur between two low-level processes running on these machines.

- As discussed in Section 3.5, the separation of the two channels allows action to be initiated at one point in the network, but cause bulk data to be transferred between two other points.

What exactly is bulk data? The answer to this question depends on the specific VICE subsystem in question. In the case of the File System, the unit of bulk-transfer is an entire file. Whenever a remote procedure call incorporates a file as one of its parameters, the actual transfer of bytes in this file is done over the bulk-transfer channel. For a Remote Disk Subsystem, on the other hand, the unit of bulk-transfer is a page. Transfers on the bulk-transfer channel are therefore page transfers between caller and callee.[2]

Figure 1 shows a typical connection. Each channel has a ***Stub*** at its end, performing the necessary housekeeping activities for communication on that channel. The basic operations on the request channel are the sending of an RPC request and the receiving of an RPC response. On the bulk-transfer channel the basic operations are the sending or receiving of a bulk unit.

It is important to note the following asymmetry in the behaviour of the stubs:

[2]Can more than one type of bulk unit be associated with a connection? In theory there is no reason why this shouldn't be possible, since the bulk-transfer protocols may disambiguate between the different kinds of bulk units. For simplicity, however, I believe that we should restrict ourselves to only one kind of bulk unit per connection. Cases where there is a perceived need to relax this constraint are probably those where there is a muddling of abstractions in the VICE Subsystem in question.

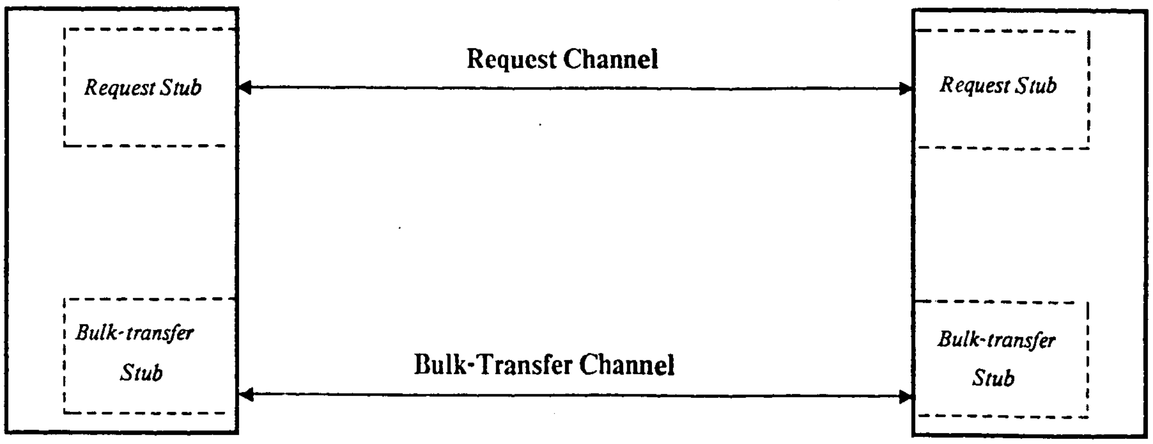

Figure 1: A Request/Bulk-Transfer Channel Pair

- On the request channel, the workstation stub is always the initiator of an RPC sequence. In other words, the workstation end plays the role of a client while the VICE end plays the role of a server.

- These roles are reversed on the bulk-transfer channel. The VICE stub, behaving like a client, initiates the sending or receiving of a bulk unit. The workstation stub acts as a server, responding to the requests presented to it by the VICE stub.

Within this general communication framework a number of specific questions remain to be answered:

- What is the format of data on the request channel?

- How is activity on the request and bulk-transfer channels coördinated?

- How is a connection created? How does a request/bulk-transfer channel pair come into being?

- What is the protocol on the bulk-transfer channel?

- How much interdependency is there between the request channel stubs and the the bulk-transfer channel stubs?

- How is security handled?

We explore the answers to these and related questions in the following sections of this paper.

3.2 Data Formats

The VICE Communication Protocol will be used by a variety of workstations to obtain services from VICE. To allow interchange of information between such systems, it is necessary to specify a common data format for use in remote procedure calls.

The set of VICE data types is described in a table of entries, one entry per data type. Each entry has the following fields:

Name	of the data type, such as INTEGER, STRING, etc.
Size	of the data type: FIXED or VARIABLE. For fixed length data types, the actual size is specified in this field. For variable length data types, the first part of the data is an INTEGER specifying the remaining length in bytes.
Prose description	giving further information about the data type. This is intended only for human understanding, in contrast to the two previous fields which may be easily interpreted by programs. Initially, all the interesting semantic and representational details about a data type will specified in this description. Later, we may be able to better categorize the data types and factor out further characteristics in a machine-understandable fashion.

In describing these data types it is important to note that "left-to-right" corresponds to the logical transmission order. In other words, a sequence of bytes composing a data type is transmitted left to right, without any padding. The bits within each byte are also logically transmitted left to right. If any deviations from this convention are necessary for actual transmission, it is the responsibility of the transport layer software to revert the data stream to the canonical order.

In some cases, conversions may be necessary between VICE data types and the data types use by an application program. For example, strings in C are represented as a null-terminated sequence of bytes without any length field. Another example is the byte-swapped representation of integers on machines such as the PDP-11. In all such cases it is the responsibility of the application program to perform the necessary conversions. It is probable that a set of library routines performing these conversions will be created for each programming language on each type of workstation.

Appendix I specifies the set of VICE data types.

3.3 Bulk-Unit Descriptors

A ***Bulk-Unit Descriptor*** is a VICE data type that functions as a placeholder for a bulk-unit in a remote procedure call parameter list. The contents of such a descriptor are passed, uninterpreted, by the callee VICE subsystem to the appropriate bulk-transfer stub. To a first approximation, a bulk-unit descriptor contains information that must be uttered to the bulk-transfer stub at the workstation in order to effect the transfer of a bulk-unit. Alternatively, a bulk-unit descriptor may be viewed as the information needed define a complete execution instance of the corresponding bulk-transfer protocol. The latter view is probably more appropriate when the descriptor contains information pertinent to issues such as flow control and encryption.

The use of bulk-unit descriptors permits a decoupling of request and bulk-transfer channels. Typically, the dependency between these two channels will be as follows:

- The bulk-transfer channel has no dependency on the request channel.
- The callee has to know how large a bulk-unit descriptor is, and to pass descriptors to its bulk-transfer stub.
- The callee has to be able to communicate the identity of bulk units to its bulk-transfer stub.
- The caller has to be able construct a bulk-unit descriptor and to communicate information such as encryption keys with its bulk-transfer stub. There is thus a greater degree of interdependency at the caller end than the callee.

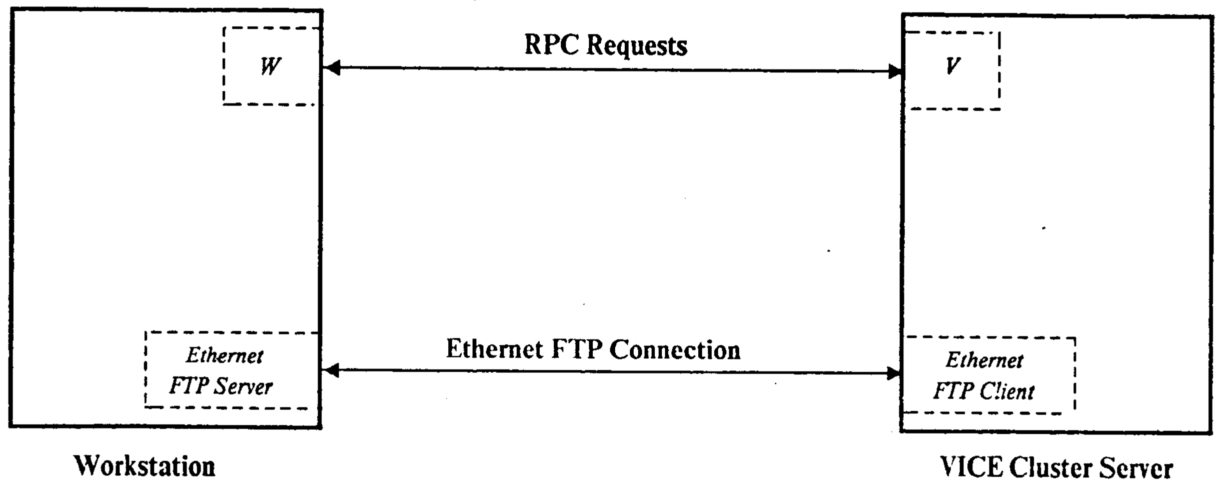

Figure 2: A File Fetch Remote Procedure Call

The hypothetical example in Figure 2 will probably clarify matters. In this figure, the local file system, *W*, on a workstation wishes to communicate with a VICE File Server, *V*. Both *W* and *V* are implemented on top of Unix, with a regular Unix file system underneath them. The bulk transfer mechanism is a simple, unencrypted, Ethernet FTP: an FTP server runs on the workstation, and the VICE node can initiate FTP requests. *W* wishes to fetch the VICE file "/user/bovik/thesis/FlatEarth.mss" and store it under the name "/cache/file019"in its local Unix file system. The FTP server on the workstation recognizes a userid "guest" with password "anonymous" for transfers into the directory "/cache".

The sequence of events in such an RPC are as follows:

1. *W* will first construct a BULKUNITDESCRIPTOR of variant type TCP-FTP-DESCRIPTOR[3] containing its own FTP server's network address, and the information "/cache/file019", "guest", and "anonymous" in the appropriate descriptor fields.

2. It will then construct an RPC message in which the opcode is **Fetch**, the first argument is of type STRING and value "/user/bovik/thesis/FlatEarth.mss", and the second argument is of type BULKUNITDESCRIPTOR and its value is the recently-constructed descriptor.

3. On receiving this RPC message, *V* examines it and discovers that it is a **Fetch** request. It knows the number and types of arguments for this call, by a table lookup.

4. *V* now executes the RPC, performing access rights checks and other related activities. It discovers that the requested file is present under the Unix name "/vicecache/xxx023".

5. It therefore requests its FTP stub to perform a **SendFile** operation, passing it the descriptor and the string "vicecache/xxx023".

6. The FTP stub initiates the file transfer, identifying itself to the remote workstation's FTP server as "guest", and copying the local file "/vicecache/xxx023" to the remote file "/cache/file019". On completion, the stub returns a successful completion code to *V*.

7. *V* now constructs a success response, and sends it back to *W* on the request channel.

3.4 Creating and Using an RPC Connection

The RPC mechanism is built directly on top of the socket mechanism, and makes extensive use of socket primitives. The request channel is nothing more than a stream socket connection between caller and callee. The request channel stubs are fictitious, and the RPC mechanism is implemented as a set of conventions imposed on data transfers on this stream socket connection. Bulk transfer stubs, on the other hand, are nontrivial pieces of code, having well-defined interfaces to software on the machines they run on. Bulk transfer channels are also socket-based, but may use datagram or stream sockets.

An RPC connection between a workstation process, *W*, and a VICE subsystem process, *V*, is created in the following manner:

- *W* first looks up the string name of *V* and obtains its socket address. There are already existing socket primitives to perform this function.

- *W* creates a stream socket and connects it to *V*. The socket mechanism handles all the necessary housekeeping such as setting up of routing tables.

- *W* sends a **Connect** RPC message on this socket. *V* examines the parameters passed to it and accepts or rejects this RPC connection.

[3] This will make sense in conjunction with Appendix I.

- Assuming *V* accepts the connection, *W* sends an **InitBulk** RPC message, identifying the bulk-transfer protocol to be used and passing along an initialization descriptor. *V* hands this descriptor to the appropriate bulk-transfer stub at its end, and responds to *W*.

- The RPC connection is now set up.

A typical use of an RPC connection is as follows:

- *W* first constructs an RPC message in an internal buffer and performs a **Write** or **Send** on the socket corresponding to the request channel. In the meantime, *V* is blocked waiting for data on the socket at its end of the request channel. On being awakened, it examines the first four bytes of data from the socket to determine the length of the RPC message and reads in that many bytes into an internal buffer.

- *V* examines the opcode, performs a table lookup, and determines the number and types of each of the parameters in the RPC message.

- *V* now performs the requested operation, using the bulk-transfer channel as necessary.

- To respond, *V* constructs an RPC message with an opcode indicating a response, marks it with the tag field of the requesting message, and performs a **Write** or **Send** on the socket corresponding to its end of the request channel.

- At some point in time, *W* does a **Read** or **Receive** from its request channel socket point and reads in the response.

This sequence of requests and response repeats many times during the life of an RPC connection.

The following steps are taken to terminate a connection:

- *W* constructs and sends *V* a **TerminateBulk** RPC message, with a termination descriptor for the bulk-transfer channel.

- *V* passes the descriptor to its bulk-transfer stub, indicating termination. It then responds to *W*.

- *W* then sends *V* a **Disconnect** request. *V* responds, indicating a successful termination and then closes its socket corresponding to the request channel.

- *W* closes its socket corresponding to the request channel. It may also have to indicate termination to its end of the bulk-transfer channel.

3.5 Action at a Distance

One consequence of the separation of an RPC connection into a request and a bulk transfer channel is that the endpoints of these two channels need not be on the same pair of network nodes.[4]

[4]It is interesting to note that a bulk-transfer channel is analogous to a DMA channel in hardware.

Such a capability is likely to be useful in supporting diskless workstations operating with ***Remote Virtual Disks*** (RVDs) provided by a separate VICE RVD subsystem.

A description of an existing RVD subsystem may be found in [1]. The design of such a subsystem in VICE is an open question at this point in time. All plausible RVD designs will support the standard operations provided by actual disks — **Read**, **Write**, **Seek**, and so on. However, these designs will also have to address issues such as responsibility for the management of free storage on the virtual disk, synchronization of accesses from multiple readers and writers, and the mapping of bytes in a file to virtual disk addresses.

For the purposes of this section we assume a stripped-down RVD design, that provides only the bare minimum of functionality. Synchronization and storage management are the responsibility of the user. The RVD primitives of interest to us are **Read** (*disk block addresses*) and **Write** (*disk block addresses*).

Based on these assumptions, Figure 3 shows how a diskless workstation may use an RVD for file transfers. The sequence of events corresponding to the situation depicted in this figure are as follows:

- A workstation process, *W*, with a page-level bulk transfer stub B_1 **Connects** to an RVD subsystem, *R*, with a page-level bulk transfer stub B_2. It obtains the necessary information from *R* to initialize its storage allocation tables.
- *W* now **Connects** to a File subsystem, *F*, indicating that it wishes to use a bulk-transfer mechanism that can transfer files to an RVD. The stub B_3 corresponds to this bulk-transfer mechanism.
- When *W* wishes to **Fetch** or **Store** a file from *F*, it first tells *R* to expect page fetches or stores on B_2. *W* then makes its file request to *F*, passing along a bulk-unit descriptor for B_3.
- At *F*'s behest, B_3 transfers the requested file as a series of disk pages to or from B_2. B_2 recognizes these pages as being part of *W*'s earlier request. On completion, *R* informs *W* that the disk read or write is over. *F* informs *W* that the file transfer has been completed.
- At any time, *W* can communicate with *R* and cause pages to be transferred between B_1 and B_2.

This discussion is inevitably sketchy at this point, since the design of the RVD subsystem is currently unspecified. The important message of this section is that the proposed communication scheme permits remote data transfers to take place, without the participants in the transfer being aware that this is a remote transfer.

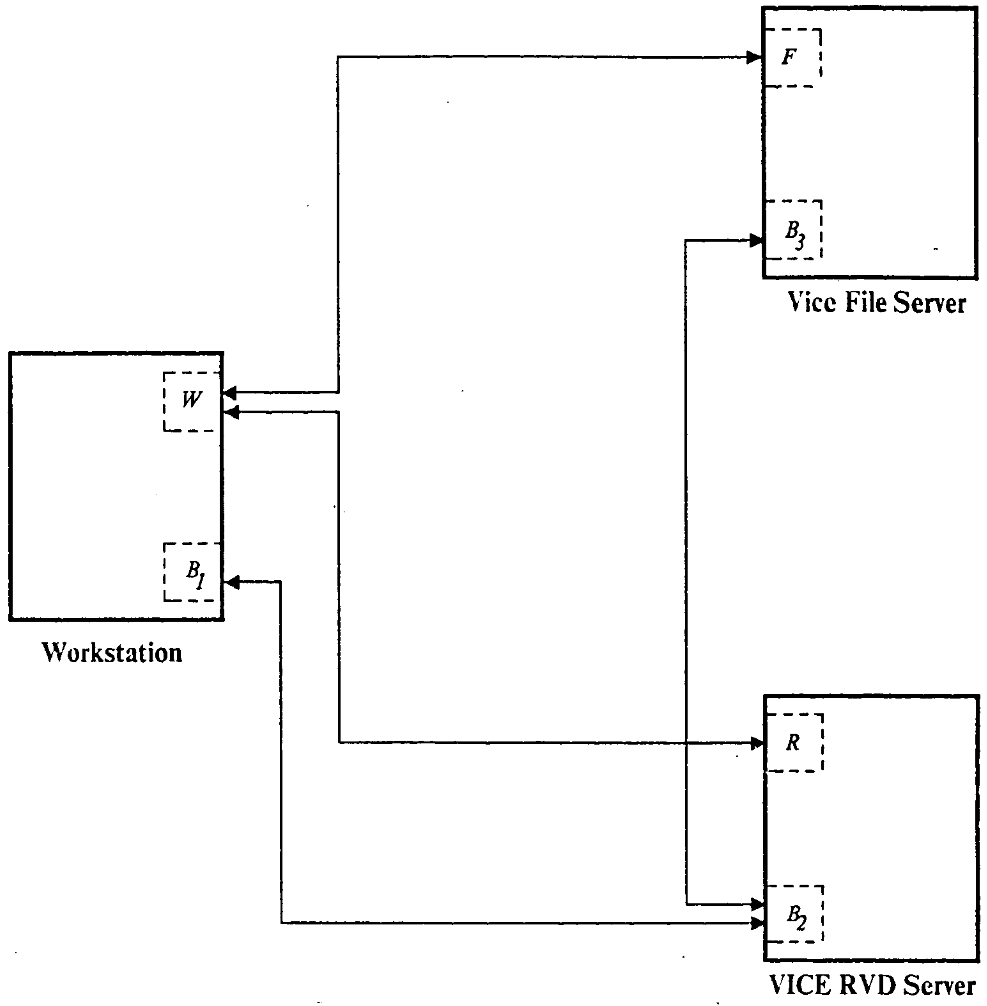

Figure 3: File Transfer to a Remote Virtual Disk

3.6 Security on RPC Connections

An extensive discussion of security in high-level protocols can be found in [6]. The design being discussed here is based on the socket mechanism, but the latter provides very little inherent security. Consequently, all security measures have to be applied end-to-end. Fortuitously, there seems to be strong evidence that end-to-end measures provide a greater degree of security than lower-level measures [6].

In the context of this design, the security concerns can be summarized as follows:

- How can a subsystem confirm the identity of the originator of an RPC **Connect** request?
- Once the identity of a requestor is confirmed, how can RPC messages on the request channel be protected against threats?

- How can a bulk-transfer server validate requests from a bulk-transfer client?
- How can data transmissions on the bulk-transfer channel be protected against threats?

The threats of most concern to us are:

- Masquerading the identity of a user's process.
- Revealing information on the request and bulk-transfer channels.
- Undetected modification of data transmissions on request and bulk-transfer channels.

Since VICE subsystems are implemented on trusted and physically secure network nodes by trustworthy(!!) programmers, we ignore Trojan horse threats. We also choose to ignore traffic analysis threats on data transmissions. Encryption is the basis for countermeasures to all the other threats.

The authenticity of a **Connect** request is established by a handshaking procedure described in [7]. One of the outcomes of this authentication procedure is the establishment of encryption keys for transmissions on the request channel. All such encrypted transmissions are protected against exposure and undetected modification.

Security on a bulk-transfer channel is derived from the already-established security on the corresponding request channel. A suspicious bulk-transfer protocol would use encrypted transmissions, with the request channel acting as a key-distribution channel. A requesting process on a workstation would pass encryption keys directly to its bulk-transfer stub; the remote bulk-transfer stub would receive its key from the bulk-unit descriptor passed to it via the request channel.

A number of details still remain to be worked in this scheme. To some extent they cannot be fully specified until the bulk transfer protocols are defined.

I. VICE Data Types

INTEGER	4 bytes 32-bit integer in 2's complement form. Representation is IBM-370 style, with bit 0 to the left and bit 31 to the right. Bit 0 is the sign bit.
TIMESTAMP	8 bytes We need to define a VICE-wide time representation. 64-bit resolution ought to be plenty. Something along the lines of "microseconds since 00:00 Jan 1 1900 GMT" should be reasonable.
STRING	Variable Sequence of bytes in ASCII. Maximum length is limited to 2^{31}-1 bytes in theory, but is likely to be far less in practice.

BULKUNITDESCRIPTOR

Variable

A variant type, the specific variant being determined by the bulk-transfer protocol specified when the connection is established. Contents are passed along uninterpreted by the VICE Subsystems to the appropriate bulk transfer mechanism.

Hypothetical examples of BULKUNITDESCRIPTOR variants are:

- TCP-FTP - DESCRIPTOR
  Descriptor for a file transfer using an IP/TCP bulk transfer protocol.

- LU6-FTP - DESCRIPTOR
  Descriptor for a file transfer on a bulk transfer server using LU6.2.

- PAGEFTP - DESCRIPTOR
  Descriptor for a file transfer using a bulk transfer server that knows that it is talking to a remote disk server. In addition to file information, this descriptor will contain information on disk block allocation.

- PAGE - DESCRIPTOR
  Descriptor for a page transfer bulk server.

II. The RPC Message Format

The initial part of a message is symmetric in both directions, and always consists of fields of type INTEGER, specifying the following:

Length	of message contents in bytes, excluding this field.
Protocol Version Number	
Tag	To uniquely identify this message on this connection. The caller always generates odd-numbered tags, and the callee even-numbered ones.
OpCode1	The opcode assignment is unique over all VICE subsystems, and consists of a pair of integers. The first integer identifies the callee VICE subsystem.
OpCode2	This is the second part of the opcode, and identifies an operation meaningful to the callee VICE subsystem. Appendix III identifies certain operations which are meaningful to all VICE subsystems.

This is followed by a list of parameters of types appropriate to the opcode. On a response, these parameters are the values being returned.[5]

[5]No parameters of BulkUnitDescriptor will be returned; remember that all bulk transfers are handled by the callee stub.

III. Common VICE Subsystem RPC Opcodes

Every VICE Subsystem recognizes the following RPC opcodes:

Connect	(. . . .) The parameter list is specific to each subsystem.
InitBulk	(BulkProtocol: INTEGER, InitialDesc: BULKUNITDESCRIPTOR) Identifies the bulk-transfer protocol to be used, and specifies a descriptor providing initialization information.
TerminateBulk	(FinalDesc: BULKUNITDESCRIPTOR) Specifies a descriptor that will cause termination operations to be performed on the bulk-transfer channel.
Disconnect	(. . . .) The list of parameters is subsystem-specific.
Response	(RequestTag: INTEGER, ...) This opcode is used by VICE subsystems to indicate a response to an RPC message. The first parameter to identifies the RPC message that this is a response to. The other parameters are subsystem- and operation-specific.

A **Connect** followed by an **InitBulk** establishes a connection, while a **TerminateBulk** followed by a **Disconnect** destroys it.

IV. The VICE RPC Database

If you got this far, you got further than I did!!

References

[1] Greenwald, M.
Remote Virtual Disk(RVD) Protocol Specification (Version 4).
Technical Report, Laboratory for Computer Science, Massachusetts Insitute of Technology, November, 1983.

[2] *Systems Network Architecture: Transaction Programmer's Manual for LU Type 6.2*
IBM Corp., Research Triangle Park, NC, 1982.

[3] Information Sciences Institute.
Internet Protocol: DARPA Internet Program Protocol Specification.
Technical Report RFC791, University of Southern California, Marina del Rey,CA, September, 1981.

[4] Information Sciences Institute.
Internet Control Message Protocol: DARPA Internet Program Protocol Specification.
Technical Report RFC792, University of Southern California, Marina del Rey,CA, September, 1981.

[5] Information Sciences Institute.
Transmission Control Protocol: DARPA Internet Program Protocol Specification.
Technical Report RFC793, University of Southern California, Marina del Rey,CA, September, 1981.

[6] Kent, ?. and Voydock, ?
Security in Higher-Level Protocols.
Computing Surveys 15(?), ?, 1983.

[7] King, D.
Authorization and Accounting.
1983.
Information Technology Center, Carnegie-Mellon University.

[8] Leffler, S.J., Fabry, R.S., and Joy, W.N.
System Internals Manual for the Sun Unix System. : Interprocess Communication Primer.
Sun Microsystems Inc., Mountain View, CA, 1983, .

[9] Leffler, S.J., Joy, W.N., and Fabry, R.S.
System Internals Manual for the Sun Unix System. : Networking Implementation Notes.
Sun Microsystems Inc., Mountain View, CA, 1983, .

[10] Satyanarayanan, M.
LU6.2 on Unix 4.2BSD.
October 1983.
Internal memorandum, Information Technology Center, Carnegie-Mellon University.