NOVEMBER 1988

WRL
Technical Note TN-7

TCP/IP PrintServer: Server Architecture and Implementation

Christopher A. Kent

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

TCP/IP PrintServer
Server Architecture
and Implementation

Christopher A. Kent

November, 1988

Abstract

The TCP/IP PrintServer is a printer that uses Internet protocols to communicate with its clients. This document describes the internal operation and implementation of the printer-resident software.

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

1. Introduction

The TCP/IP PrintServer is a freestanding Ethernet-connected print server that uses TCP/IP protocols to receive files for printing. This document describes the internals of the PrintServer and the mechanisms used to process print jobs and interact with clients. The reader is assumed to be familiar with the VAXELN mechanisms for job and process management, events, and communication.

The software installation procedures are described in a companion document entitled *TCP/IP PrintServer: BSD Unix Client Interface* [9]. The TCP-based protocols by which clients communicate with the server are described in a companion document entitled *TCP/IP PrintServer: PrintServer protocol* [10]. TCP is one of a family of protocols collectively referred to as "Internet protocols"; they are usually referred to collectively as "TCP/IP". TCP stands for "Transmission Control Protocol" [6] and IP stands for "Internet Protocol" [7]. The interface between the server and the PostScript interpreter is described in a document entitled *Distributed Printing Services V2.0/Controller Software Interface Specification* [5]. It is briefly summarized in section 2.

1.1. Clients, management clients, and console clients

In normal use, the PrintServer receives connections from client hosts, prints the files that it receives over those connections, and closes the connection. The host computers that connect to the PrintServer are called *clients*.

There are three different reasons that a client might connect to a PrintServer, and therefore three different kinds of client connections[1]:

- If a host connects to the PrintServer for the purpose of printing a file, then it is called an *ordinary client*.
- If a host connects to the PrintServer for the purpose of providing administrative services to it, then the host is called a *management client*.
- If a host connects to the PrintServer for the purpose of providing a remote console display to a user, then the host is called a *console client*.

Ordinary clients, and the protocols by which they communicate with the PrintServer, are described in the companion *BSD Unix Client Interface* document.

When a management client connects to the PrintServer, it sends a list of the services that it is willing to provide. When the PrintServer needs a service, it checks each of the connected management clients, and requests the service from the first one that it finds that offers that service. If none of the connected clients offers that service, then the request is abandoned and some suitable corrective action is taken. Other than during the interval between a reboot and the first management client connection, the PrintServer will never wait for a management service to be-

[1] Note that from the point of view of the client software, as described in the *Client Interface* document, there is fourth kind called a *remote client*, but remote clients never connect to the server and are therefore not relevant to server management issues.

come available.[2] In an ordinary installation, there will be one management client that provides all services, and possibly a second for backup to make sure that accounting information is never lost. The ability to factor out the different services and offer them from different places is one that will not be needed in any but the most complex of installations.

It is perfectly correct for a single host to be an ordinary client and a management client all at the same time. It is also quite correct for a host to be an ordinary client for one PrintServer while being a management client for another.

1.2. The protocol

In the era of line printers, one communicated with a printer by sending it data, which it printed. No protocol was needed beyond link-level issues such as framing and flow control. Modern high-speed PostScript printers are connected via a Local Area Network (LAN) such as Ethernet. To print a file on such a print server, one must cope with network connections, access control, error recovery, remote accounting, and support for various features of the PostScript language such as file I/O.

The PrintServer protocol [10] is an Internet protocol that is built on top of TCP. TCP provides full-duplex, reliable, flow-controlled byte stream communication between two nodes on a network. The PrintServer protocol is a record-oriented protocol that uses TCP streams as its transport mechanism.

Printing is done in units called *print sessions*, or simply *sessions*. A session is an uninterrupted connection from a client to a PrintServer. A client opens a session, then for each file to be printed, it starts a *job* within that session, prints the file, and ends the job. At the end of the session, the client releases the network connection, and another client is free to print.

1.3. Goals and non-goals

The IP-based PrintServer is intended to make the PrintServer product line usable from any computer that can speak IP, with an emphasis on Ultrix and UNIX systems. In particular, the project goals are:

- Provide PrintServer access to any system that speaks IP, especially UNIX-compatible systems.
- Provide 40 page per minute throughput on large print jobs.
- Use the same Adobe-supplied modules, for the PostScript interpreter and LNV11 driver, as the LAPS-based system.
- Provide a printing protocol that is simpler than LAPS. This protocol should be placed in the public domain, and be easy to interface to the existing UNIX `lpd` line printer system.

[2] If the printer is configured for reliable accounting and no management client is offering accounting services, the PrintServer will wait until the accounting service becomes available.

- Use TCP/IP as the base communication protocol family.
- Provide network access control.
- Provide centralized per-job accounting.
- Provide centralized error and event logging.
- Provide a mechanism for remote storage of fonts.
- Provide a console in a style familiar to UNIX users (command based instead of menu based).
- Provide remote console ("remote server management") facilities.
- Provide a configuration mechanism that is not VMS-specific and can be easily applied to the needs of UNIX.
- Allow up to 16 client connections at one time.

The following are specific project non-goals:

- Provide support for any PrintServer other than the LPS40.
- Provide all VMS-based PrintServer-provided functions in a UNIX environment. This includes but is not limited to server management.
- Use LAPS as a print/client protocol.
- Internationalization.

2. Overview

Four pieces of software are used to print on the PrintServer:

`lpr` — There is a special back end to the standard Unix line printer spooler that uses the PrintServer protocol to transmit print jobs to the PrintServer.

`lpad` — One or more machines on the network provide support services by running a management client, `lpad`.

`lprc` — This program provides remote console services.

server — The software that runs on the PrintServer itself.

The server software is an application that runs on the MicroVAX II inside the PrintServer. It is written using the VAXELN operating system, and consists of several VAXELN jobs, as shown in Figure 1.

One job handles all client interactions with the PrintServer. This is known as the *server job* and is the subject of this report.

Two jobs cooperate to provide the PostScript interpreter and drive the print engine hardware. These are known jointly as "the controller". The code for these jobs was obtained from Adobe Systems, Inc.

The interface to the controller is via four channels, implemented as VAXELN (reliable) circuits. Communication is in terms of Packets, each of which is sent in a single VAXELN

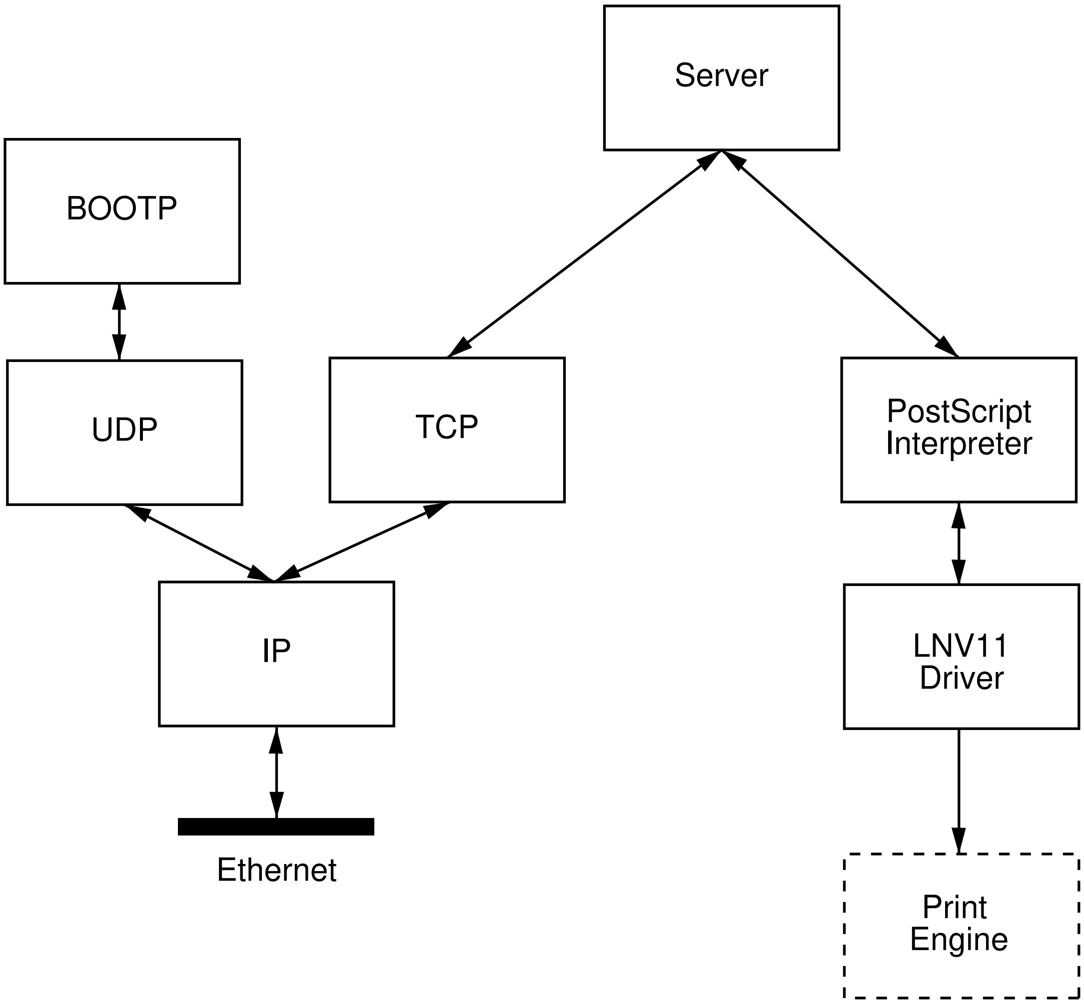

Figure 1: Jobs in the PrintServer application

`MESSAGE`. A Packet consists of a header and some data. The header contains job boundary markers and identifiers, as well as a description of the data portion of the Packet. The four channels are:

Data	The controller treats messages arriving over the data channel as Packets containing data to be consumed by the PostScript interpreter via its standard input stream. The sequence of Packets from the one with its "start-of-job" bit set to the one with its "end-of-job" bit set (inclusive) constitute one PostScript file, which is ordinarily executed as one PostScript job.
Status	The controller sends controller *status blocks* to the status channel in response to specific events detected by the PostScript interpreter. These events are: PostScript language-level errors, job abort, and exiting the server save/restore context. Characters written to the standard output file result in the sending of user status blocks over the status channel. When the standard output file is closed (which ordinarily happens only at the end of each PostScript job), the controller sends a special controller status block as an end-of-file marker for the

standard output file of the current job. This status block is not sent until the last page of the job has been delivered to the output stacker.

The use and disposition of status blocks are discussed more fully in Section 2.1.2.

Resource — Resource status blocks are generated by execution of PostScript operators that require delivery of resources via the resource channel. The two resource requests defined at present are font fault and password check.

When the PostScript **findfont** procedure is executed for a font that is not present in the PostScript VM, it generates a resource status block that requests that the font be loaded via the resource channel. At present, this request is always denied by sending an error Packet down the resource channel.

When the **checkpassword** operator is executed, it generates a resource status block requesting that the supplied password be checked. It then awaits a success or failure indication on the resource channel.

Control — The controller reads Packets asynchronously from the control channel. When it receives one, it pays attention only to the length in the header and the body. It treats the body as a sequence of control blocks, which it acts upon immediately. A control block consists of a *type* and a *size*.

In response to a "show status" request, the controller sends a controller status block and a printer status block containing the current status of each.

In response to an "abort" request, the controller forces execution of the PostScript **interrupt** error from **errordict** [1]. Ordinarily, this is delayed until the next object is about to be executed by the PostScript interpreter; however, if the interpreter is blocked waiting for data to arrive from the data or resource channel, the wait is aborted and the **interrupt** is executed immediately.

Three jobs cooperate to provide network services: one for the TCP protocol, one for the IP protocol, and one for the BOOTP protocol, which is used to determine the PrintServer's IP address at boot time [3]. The code for these jobs was obtained from Process Software, Inc.

The TCP/IP interface was designed to present the same interface as the standard DECnet facilities for reliable circuit connections [4]. Additional management functions are provided by sending control messages to the IP and TCP control **`PORT`**s [8].

2.1. Processes within the server job

The server job is made up of four groups of processes: the main process, the status collector, the PrintServer protocol handler, and the console. Error logging, accounting and remote file access are provided by a module that uses management connections, but do not exist as separate processes. (See Figure 2.)

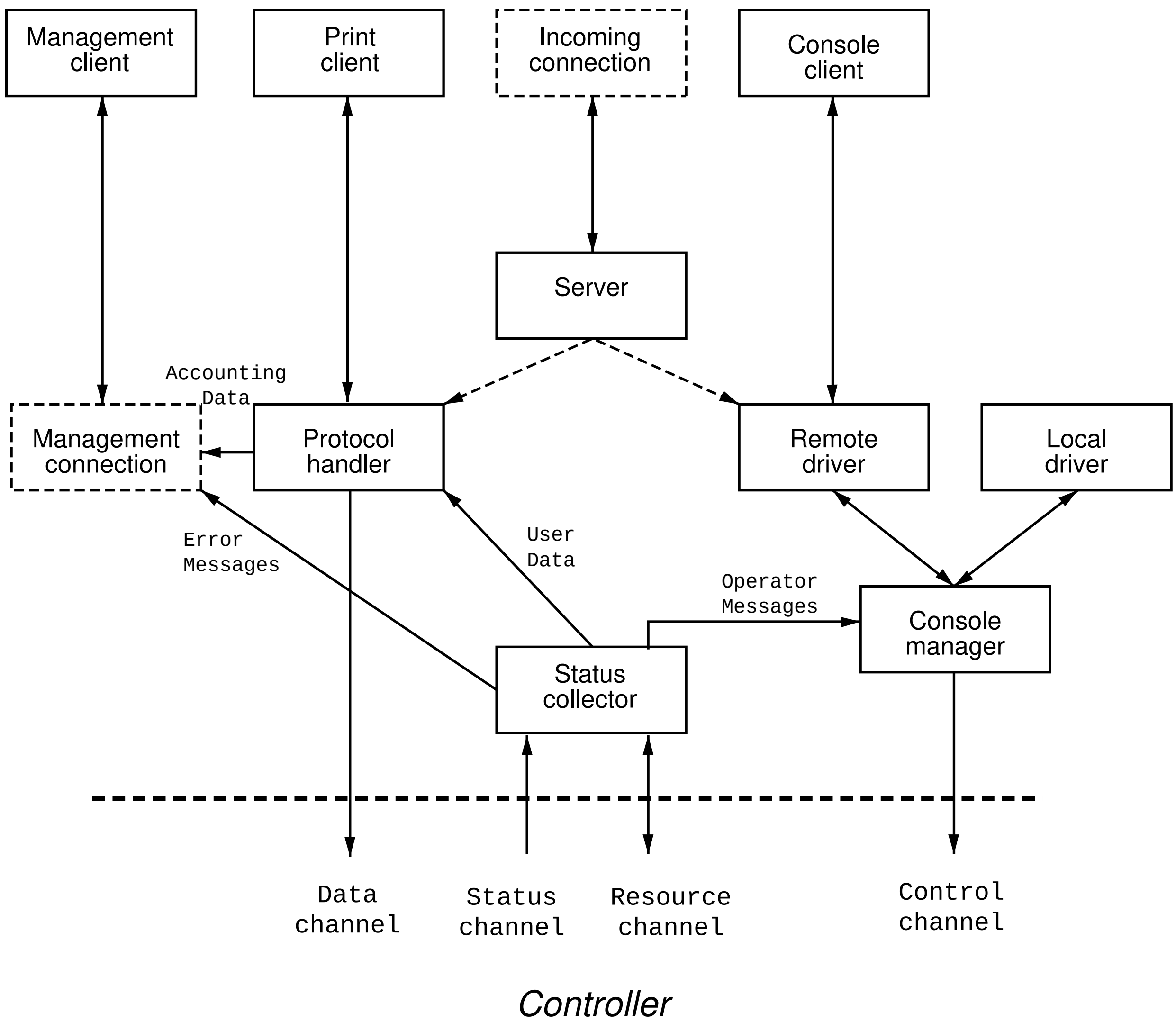

Figure 2: Processes in the server job

2.1.1. Main server process

The main server process is the first process started in the server job. It begins by starting the PostScript interpreter job and the LNV11 driver job. Next, data structures are initialized and the other process groups are started.

Finally, it goes into an infinite loop, accepting incoming connection requests and dispatching them to the protocol handler.

2.1.2. Status collector

The controller communicates events in the PostScript interpreter and print engine by sending status blocks over the status channel. A status block consists of a class (one of Printer, Resource, Controller or User), a condition record, and some class-specific fields. A condition record contains a set of recipients, an action code, a message code, and zero or more arguments.

The status collector receives status blocks from the status channel, interprets them, and dispatches them. It looks up the message text associated with the message code and uses that and the argument list to form a message string. If the recipient set does not indicate that the message is only for internal use, the message string is sent to the indicated recipients. (Possible recipients are the user, the error logger, and the console log.)

The action code indicates what action, if any, should be taken with respect to the job. This condition record may be for information only, may pause or resume the job, or may indicate that the job is complete or will be aborted.

2.1.3. PrintServer protocol handler

The PrintServer protocol handler is invoked as a process that is started for each connection that is accepted by the main server process. The client's first protocol record indicates whether this connection is to transmit a print job or jobs, establish a management connection, or to start a remote console.

If this is a management or console connection, the protocol handler creates a new process with the appropriate handler, passes the connection to the new process, and exits.

The job handler consists of two processes: the protocol parser and an abort monitor.

The protocol parser either accepts or rejects "start session" requests depending on the number of clients already connected, the system state, and the access control list. If a connection is rejected, the parser sends a message containing a description of the reason and breaks the connection. If the connection is accepted, ensuing protocol records are interpreted and passed to the PostScript interpreter as appropriate.

The session is composed of one or more PostScript jobs. For each job in the session, the protocol parser informs the controller that a job is starting, relays the client's uninterpreted print data, informs the controller that the job has ended, and waits for a synchronization message from the controller (via the status collector) that indicates that the last sheet of the job has been successfully printed. Should the connection be broken in the middle of a job or session, the protocol parser is sure to send the controller an end of job indication and allow the controller to clean up properly in preparation for the next job.

The abort monitor watches the TCP connection for "urgent" data. The client sends job abort requests as urgent data; the abort monitor is responsible for receiving and carrying out these requests while the protocol parser is busy doing other things[3]. If an abort request is found, the monitor sends a control channel abort request to the controller and signals the protocol parser to quit. The protocol parser will see both the quit signal and the abort request; it will heed the quit signal but ignore the abort request.

If the protocol parser happens to see the abort request before the abort monitor does, it sends a control channel abort request to PostScript and begins to quit. In this case, the abort monitor will not issue an abort request.

[3]For example, waiting for the PostScript interpreter to finish an infinite loop.

2.1.4. Console

The console display is divided into three parts: the banner, shell, and log. The banner shows a short summary of the printer's current status. The log is an append-only record of messages intended for an operator's attention, including both print job and hardware status. The shell is the area where an operator can use a line-oriented interface to inquire into and alter the full state of the printer.

The console code handles both local and remote consoles. There is one local console active at all times; in the standard configuration, there can be up to 10 remote consoles active simultaneously.

2.2. System state

The status collector maintains state containing the most recent condition records having to do with the PostScript interpreter and the print engine, as well as other configuration information. It exports a set of interfaces for querying and updating this state.

The job queue is a linked list of records, one per active job. The queue is served in FIFO order. Each job record contains job identification information, the network connection handle, and three **`EVENT`**s: **proceed**, **done**, and **abort**.

After a job has been added to the job queue, it waits for the **proceed `EVENT`**. The job manager signals **proceed** when the job has reached the head of the queue.

When a job has delivered all its data to the controller, it waits for the **done** and **abort `EVENT`**s. The controller sends a condition record to the status collector when the last sheet has been successfully delivered to the output stacker; the status collector then signals **done**. If an operator or the client aborts the job during this period, the effect is to signal **abort**.

The console subsystem maintains a state record for each active console. This state contains the input buffer, state about the current mode of the console (e.g., repeat mode and privileges), and a set of input/output *methods*. The methods provide a standard interface for performing console I/O, hiding the implementation differences for local and remote consoles. All console commands are implemented in terms of these methods.

The server maintains a record of the currently active management connections and the services they offer. The server uses these connections to send accounting and error log records, and to perform remote procedure calls [2] for file and time services. If a connection fails, the server notices and removes the associated record.

Protocol records are sent over a TCP connection. TCP provides to its clients the illusion of a seamless byte stream, but in reality, the stream is broken up into *segments*. Each of these segments is sent in a single IP packet. VAXELN doesn't embody the concept of a byte stream network connection -- the incoming data is always broken up into *messages*. This means that, depending on network parameters, flow control, buffering, and the phase of the moon, several protocol records may appear in one message, or one protocol record may be broken across two or more messages.

A *Frame* provides an abstract interface to the network that reconstructs protocol records regardless of these details. Routines in the server call Frame utilities to read (or write) a single protocol record from (or to) the network, and the utility routine handles all details of buffering and reconstruction.

3. Operation

So far, the server looks like a collection of facilities. To see how they interact, we now consider the lifetimes of a print session, a console, and a management connection.

3.1. A print session

A print session begins when a client tries to establish a TCP connection to the PrintServer's service port. The master server job is waiting for a connection request; when it receives the request, it accepts the circuit, specifying that a small receive window should be used. The server now creates a child process, running the PrintServer protocol handler.

The protocol handler allocates and initializes a Frame, and loops receiving and processing incoming procotol records.

Client sends	Server response	Comments
SSN		*begin a session*

The child tries to allocate a job queue entry. If the queue is full, if job acceptance is disabled, or if the PrintServer hasn't been contacted by a management client to get its configuration information, the child sends a `NAK`, closes the connection and exits. The `NAK` contains a verbose indication of why the job was refused.

	NAK	*refuse it*

If the child accepts the job, it records the session-specific information in the job queue entry, attaches the job to the tail of the job queue, sets up the abort monitor, sends a `REPLY` with job identification, and logs a message on the console. If the job ends up at the head of the job queue, attaching the job has the side effect of signalling the job's **proceed `EVENT`**.

	REPLY	*acknowledge it*
INFO		*identify the coming print job*

The `INFO` record contains information specific to the first print job in the session. The job queue entry and the console banners are updated.

SOJ		*start a print job*

The client is about to start sending data. At this point, the child blocks until the job is at the head of the job queue. This is accomplished by waiting until the **proceed `EVENT`** is signalled. If the job is already at the head of the queue (because all other jobs have completed, or it was attached at the head of the queue), **proceed** will already be signalled.

If the PrintServer is in single job mode, the operator must manually release the job. In this case, **proceed** is cleared just before it is waited on, and the operator's intervention signals it again.

The child also waits on the **abort `EVENT`**, to allow the client or an operator to abort the job before it starts running. If this occurs, the child sends a `KILL` packet to the client and detaches the job from the queue.

	`KILL`	*notify the client that the job is dead*

When **proceed** is signalled, the child opens the TCP receive window to allow the client to send data. It then zeroes the accounting counters, and clears the "Start of job sent to PostScript" state bit (SOJ).

`DATA`		*send the break page*
`...`		
`DATA`		
`EOJ`		*end of break page*

`DATA` messages are repackaged into Packets and sent directly to the controller. If SOJ is clear, the "start of job" bit is set in the Packet when it's sent to the controller, and SOJ is set. The connection to the controller is flow controlled by ELN, so the protocol interpreter will block when the controller gets behind.

When an `EOJ` packet is sent, the "end of job" bit is set in the next Packet sent to the controller. This signals the controller to send a synchronization condition record when the last sheet has been delivered to the output stacker. The job waits on the **done `EVENT`**, which will be signalled by the status collector when this condition record arrives. At that point, the child sends an accounting record to the client (both as a `REPLY` and on all management connections that have advertised the accounting service).

	`REPLY`	*Return page count*
`INFO`		*identify the coming print job*
`SOJ`		*start a print job*
`DATA`		*send file 1*
`...`		
`DATA`		
`EOJ`		
	`REPLY`	*Return page count*

Further print jobs follow the same pattern. The child does not give up the session's position in the job queue; as long as the client maintains the connection, it controls the PrintServer.

`WAIT`		*wait for all error messages*
	`REPLY`	*announce the end*

When the client is done with all its jobs, it sends a `WAIT`. At this point, the child makes sure that the current PostScript job has seen an "end of job" and waits for the **done `EVENT`**. This is usually an empty wait, since the child has just waited for the same **`EVENT`** for the last `EOJ` message. At this time, the child relinquishes the session's position in the job queue and signals next job's **proceed `EVENT`**. The child doesn't exit until the TCP connection is closed by the client.

KILL		*abort the job*
	REPLY	*send accounting info for the aborted job*

If the client wishes to abort the current job, it sends a `KILL` record as urgent data. If the child is blocked sending to the controller, the abort monitor sends a control Packet to abort the PostScript job. If the child is not blocked, it will see the `KILL` first and abort the job. (See section 2.1.3.)

In either case, the `KILL` is treated much like a `WAIT`: the child waits for the **done** **`EVENT`**, sends an accounting record for the aborted job, and removes the session from the job queue.

The client may break the TCP connection at any time. If the child is not in a session, it kills the abort monitor and exits. If it is in the middle of a session, it aborts the current job and removes the session from the job queue first.

3.2. The consoles

The console subsystem is designed to share a great deal of code between the local and remote cases. All routines that deal with input or output do so via the uniform I/O interface provided by the appropriate methods. The local methods deal in great detail with the user interface and the specifics of writing to the VT100-class terminal that is used for the local console. The remote methods do little but format information into protocol records and send them to a peer running on the client. The client is responsible for the exact layout of the remote user interface and the details of the remote terminal. The output format is general enough that both very simple and very complex remote clients can be built.

The processes that make up the console subsystem are divided into two logical parts: the logger and banner, and the individual shells.

The logger and banner processes are simple loops that wait for an event and update all currently active consoles, using the output method recorded in that console's state entry. The logger waits for a message on its **`PORT`** (`CONSOLE_LOG`) and appends it to the log area of the consoles. The banner waits for a timeout or a message on the `CONSOLE_BANNER` **`PORT`** and updates the banner line of the consoles.

Each shell consumes two processes. One loops forever, waiting for input, filling a buffer and signalling an **`EVENT`** when there is new input. The other does the work: it waits for the **`EVENT`**, reads the completed input line, calls the appropriate command procedure, and displays the output. Both local and remote consoles have this two-process structure.

Whether local or remote, the processing loop is identical. First, print the prompt and assemble a complete line of user input. The local console does its own echo, erase and kill processing; the remote client is expected to send a complete line. The table of commands is searched for the first word on the command line. If it is found, and the command may be executed with the current set of privileges, the command routine is called with a pointer to the console state entry (which contains pointers to the output methods).

If the command is repeatable and the repeat interval is non-zero, the command is called again after the repeat interval passes. This continues until the user types any character.

After the command or repeat loop exits, the prompt is printed again and the loop restarted.

A remote console client begins the same way that a print client does: it attempts to establish a TCP connection to the PrintServer's service port. The master server job accepts the request and hands the connection off to a newly-created protocol handler process.

Client sends	Server response	Comments
CSSN		*begin a console session*

First, the protocol process opens the TCP receive window. Then, it creates a new remote shell process, hands off the connection, and exits.

The shell tries to allocate a console entry. If none are unused, it sends a NAK record, closes the connection and exits.

	NAK	*refuse it*

If a console entry is available, the shell initializes it with pointers to the remote console methods and sends a REPLY record to the peer. It then disables the input timeout and requests the banner process to update all the banner lines (so the new console will have something on its banner line).

	REPLY	*acknowledge it*
	DATA 1<banner line>	*send the banner info*

Console output is destined for one of four portions of the screen: banner, shell, log, or prompt. The first byte of the record is used to indicate the portion of the screen on which the rest of the record should be drawn: 1 for the banner line, 2 for the shell, 3 for the log, and 4 for the shell prompt. A console client can use or ignore any part of the output stream. For example, a simple client to just watch the log would print DATA packets sent to 3, but ignore all others.

Now the shell prompts for input and waits for an input packet. The prompt is treated specially so the peer can notice the length for erase and kill processing.

	DATA 4lps40 %	*send the prompt*
DATA jobs\n		*an input command*

When a complete input line has been received, the shell looks through the command table for a match on the input string. If a match is found, the appropriate routine is called with the console entry as an argument.

	DATA 2No jobs\n	*command output*
	DATA 4lps40 %	*another prompt*

This continues until the network connection is closed.

3.3. A management connection

Management connections exist in the server as VAXELN **PORT**s on which messages can be sent -- for the majority of their lifetime, there is no associated process. Routines in the server that need to make use of a management service call on a utility routine. The utility routine finds an appropriate management connection for the desired service and sends the required network traffic.

A management client begins the same way that a print client does: it attempts to establish a TCP connection to the PrintServer's service port. The master server job accepts the request and hands the connection off to a newly-created protocol handler process.

Client sends	Server response	Comments
MSSN		*begin a management session*

First, the protocol process opens the TCP receive window. Then, it creates a new management connection process, hands off the connection, and exits.

The new process tries to allocate a management connection entry. If none are unused, it sends a NAK record, closes the connection and exits.

	NAK	*refuse it*

If a management connection entry is available, the process copies the parameters from the `MSSN` record and notes which services the client offers. Error logging, accounting, and remote file access are the three types of management services. The process then sends a `REPLY` that echoes the services the client offers.

	REPLY	*acknowledge it*

Every time a new management client contacts the PrintServer, the PrintServer resets its idea of the current time from that client. Thus every client must offer time service; the PrintServer uses the action of asking for and receiving the time from a client to determine if the client is still alive.

	TIME	*ask the current time*
REPLY		*get it*

The server keeps track of whether or not it has been configured; that is, whether or not it has been contacted by a management client and successfully read its configuration and setup files. Configuration files are a degenerate case of general file access: requests for the files `$CONFIG` or `$SETUP` are translated by the management client into unique names for the printer. The server will not accept print sessions until it has successfully read the configuration files.

	OPEN	*open $CONFIG*
REPLY		*get a file handle*
	READ	*request a line of the file*
REPLY		*receive it*
...		
	CLOSE	*close $CONFIG*
	OPEN	*open $SETUP*
REPLY		*get a file handle*
	READ	*request a line of the file*
REPLY		*receive it*
...		
	CLOSE	*close $SETUP*

Finally, the process exits, leaving the connection entry allocated for further use. If, when a connection is used, the client has died or the connection has otherwise failed, the connection entry is deallocated and its services marked unavailable.

4. What's next?

The PrintServer40 is the first step in an evolutionary PrintServer family. While the entire future isn't clear, some pieces are already in place. Here are some coming attractions and what needs to be done to use this software on the new hardware.

4.1. PrintServer 20

The current code should work as is on a PrintServer 20, with the exception of the front panel driver. Implementing a driver for the front panel should be straightforward. Since the console code is table driven, all that is needed is an input command table and a set of output methods. The input command table should be constructed to map the character sequences generated by pressing front panel buttons into the appropriate commands. The output methods will be whatever code is required to speak to the front panel. Since neither the banner nor the log output are appropriate for the 20's small front panel, these output methods will be no-ops.

Some new commands may have to be written to implement the 20's specified front panel behavior, but this should not be very difficult.

4.2. Overlapping jobs

Currently, there is a 12 second delay between the end of one job and the start of the next. This delay is required to allow the paper pipeline to clear completely - it insures that all pages of the current job have been correctly delivered to the output stacker before accepting data for a new job.

Adobe has promised a version of PostScript that will provide a synchronization message when the processing for the current job is complete. This will allow us to overlap sending data for a new job with the finishing of the old job. It adds some risk: if a job is deleted from the client host's queue when its processing is done, but the printer crashes before all the pages are delivered, some pages will be irretrievably lost.

To take advantage of this new synchronization mechanism, the server code will have to be changed. However, the change appears to be simple. The code that synchronizes on an `EOJ` packet will wait for the new synchronization message, while the `WAIT` code will still wait for the "all pages printed" message. This will allow jobs within a session to be overlapped, but will try to print all pages from a given session safely.

5. References

[1] Adobe Systems, Incorporated.
PostScript Language Reference Manual.
Addison-Wesley, Reading, Massachusetts, 1985.

[2] Andrew D. Birell and Bruce Jay Nelson.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[3] Bill Croft and John Gilmore.
Bootstrap Protocol (BOOTP).
RFC 951, SRI-NIC, September, 1985.

[4] Digital Equipment Corporation.
VAXELN C Run-Time Library Reference Manual.
3.1 edition, Maynard, Massachusetts, 1988.

[5] HardCopy Firmware Group.
Distributed Printing Service V2.0 / Controller Software Interface Specification.
V2.0-3 edition, Digital Equipment Corporation, Maynard, Massachusetts, 1987.

[6] Jon Postel.
Transmission Control Protocol.
RFC 793, SRI-NIC, September, 1981.

[7] Jon Postel.
Internet Protocol.
RFC 791, SRI-NIC, September, 1981.

[8] Process Software Corporation.
TCP/IP VAXELN Software User's Guide.
Version 1.0 edition, Amherst, Massachusetts, 1988.

[9] Brian K. Reid.
TCP/IP PrintServer: BSD Unix Client Interface.
TN 5, Digital Equipment Corporation Western Research Laboratory, September, 1988.

[10] Brian K. Reid and Christopher A. Kent.
TCP/IP PrintServer Print Server Protocol.
TN 4, Digital Equipment Corporation Western Research Laboratory, October, 1988.

Table of Contents

List of Figures