# ARCHITECTURE AND IMPLEMENTATION OF THE INFOPAD NETWORK PROTOTYPE

by

Frederick L. Burghardt

Memorandum No. UCB/ERL M94/81

10 October 1994

# ARCHITECTURE AND IMPLEMENTATION OF THE INFOPAD NETWORK PROTOTYPE

by

Frederick L. Burghardt

Memorandum No. UCB/ERL M94/81

10 October 1994

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# ARCHITECTURE AND IMPLEMENTATION OF THE INFOPAD NETWORK PROTOTYPE

by

Frederick L. Burghardt

Memorandum No. UCB/ERL M94/81

10 October 1994

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# Architecture and Implementation of the InfoPad Network Prototype

Frederick L. Burghardt
(flb@eecs.berkeley.edu)

## Abstract

The InfoPad project at the University of California at Berkeley is developing a mobile computing environment in which the terminal device is a small and inexpensive hand-held unit containing no user programmable hardware. The device, or *pad*, employs a pen-like stylus as the primary user input tool; the pad is connected to a high-speed backbone network via radio pico-cells. The lack of a general purpose processor in the pad distinguishes InfoPad from similar efforts in an important and unique way: all processing is performed by compute nodes on the backbone network. This paper describes a distributed software system that has been designed and built specifically for InfoPad. The description is divided into two major parts: architecture and implementation. The architecture is an abstract definition that specifies how network compute resources are organized to meet InfoPad needs. It outlines the basic structure necessary to deliver data from a network based application to a mobile pad. This structure includes basic compute elements, interconnections, communication protocols, algorithms, and programming interfaces necessary to perform tasks such as pad location, data management, and connection handoff. The second major section of the paper is a detailed description of an implementation of this architecture. A working prototype is now in daily use on a general-purpose Ethernet; the prototype supports InfoPad application development and research into issues such as radio cell power management, pad relocation, management of bandwidth, and error rate detection and control. Multiple pads can be running at any given time controlled by a number of network processes running on any network compute node. Both the architecture and implementation are continually evolving, and are expected to be a useful research platform for the life of the InfoPad project.

# Contents

## List of Figures

# 1 Introduction

This paper describes the architecture and implementation of a prototype network-based software system. The system was built to support research into mobile computing in a multimedia environment. It forms a development testbed for the network portion of the InfoPad project currently underway at the University of California at Berkeley.

The system is intended to give InfoPad application programmers a "real world" environment unavailable through analytical modeling or simulation. It is also intended to allow experimentation with mobile computing issues such as connection handoff, bandwidth management, and quality of service. In addition, it serves as a proof-of-concept vehicle to test project assumptions and expectations in both quantitative and qualitative ways.

## 1.1 InfoPad Overview

The purpose of InfoPad is to develop a mobile computing environment in which the terminal device is a small and inexpensive hand-held unit containing no user programmable hardware. The device, or *pad*, employs a pen-like stylus as the primary user input tool. The pad is connected to a high-speed backbone network via radio pico-cells.[1] The lack of a general purpose processor in the pad distinguishes InfoPad from similar efforts in an important and unique way: All processing is performed by compute nodes on the backbone network, and data transferred between pad and network is "raw" in the sense that only primitive operations will be applied to data on the pad. This approach transfers many tasks normally handled by a local machine to remote processors, significantly increasing the burden placed on the network. The aggregate bandwidth of an InfoPad data channel will be 2Mbps.

The system can be viewed as a pair of subnets that connect two endpoints. One of these endpoints is the pad, and the other is an application process running on a network compute node. The pad and radio hardware form a "wireless" subnet, and the backbone network forms a "wired" subnet. The interface between these subnets is the point at which the network-side radio tranceivers connect to the backbone network.

From the viewpoint of a pad, this interface is temporary i.e. the communication channel between pad and wired subnet moves from interface to interface as the pad moves from cell to cell. This action is called *relocation* or *handoff*.

The InfoPad project is technically split into two parts. The first is concerned with the wireless subnet and the second is concerned with the wired subnet. This paper discusses the wired subnet in detail; the wireless subnet will be covered only indirectly. The prototype described here exists on the backbone network and has no specific knowledge of wireless subnet hardware elements. However, the choice of hardware will eventually have a significant impact on the network system, so developers have attempted to anticipate hardware needs and design accordingly. At this stage of InfoPad, the network architecture contains elements that are considered basic and general. Figure 1 shows a top level view of the InfoPad system.

[1] Pico-cells are about 10 meters in diameter. As an alternative, infrared communication is under consideration.

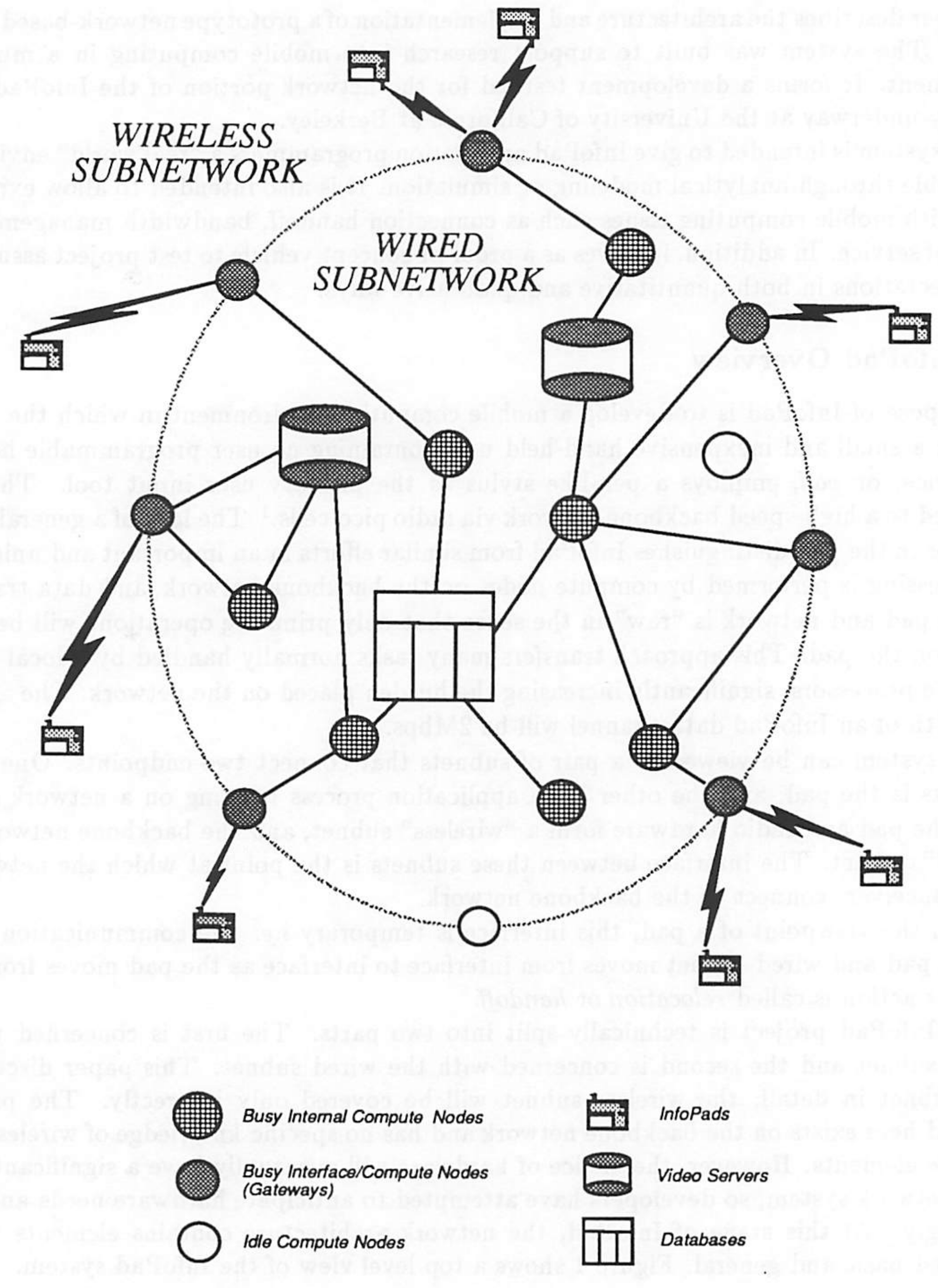

Figure 1: Top level schematic view of major InfoPad components.

## 1.2 Goals

As an early prototype in a research environment, ultimate system requirements are difficult to predict. However, given the role the prototype is expected to play, a set of important generic goals can be identified:

- **Speed:** The system must be fast in order to process multiple streams of multimedia data in real-time.
- **Efficiency:** The system must present a small overhead burden to the network compute nodes, since processor cycles consumed by the system could otherwise be used by applications.
- **Simplicity:** The system must be relatively easy to understand, easy to modify, and portable. It must not pay for functionality not required [16].
- **Reliability:** The system must not present a burden to application programmers due to frequent (or even occasional) unavailability.
- **Concurrency:** The system must be able to handle multiple independent streams of data and control at a relatively fine granularity.
- **Fairness:** The system must give equal priority to all data streams unless explicitly told otherwise by a priority based scheduling algorithm.
- **Flexibility:** The system must allow easy addition and modification of function so that it can grow with the needs of the project. Individual components must be distributed in order to take advantage of the compute power of a large network.

## 1.3 Organization

The terms *prototype* and *system* represent all elements of the network portion of InfoPad that have been implemented in some form or are expected to be implemented in the near future. This paper examines the prototype on two levels: an abstract architectural specification and an implementation of that architecture on a real network. There are aspects of the system that are beyond the scope of this paper. They will be referenced where appropriate.

The architecture describes a software entity. The discussions that follow generally assume that computation is not limited by specific technologies, and for the most part this is true. A hardware link has been developed and extensively used for performance testing, but is not considered to be part of the architecture [8].

The remainder of this paper is organized as follows: Section 2 introduces InfoPad data types and traffic characteristics. Section 3 describes the system architecture. Sections 4 and 5 examine two aspects of architecture implementation. Finally, section 6 presents some conclusions and a review of the degree to which the system meets the goals outlined above.

## 2 InfoPad Data Types and Traffic Characteristics

InfoPad defines four types of traffic: audio, video, pen, and text/graphics. Each type can be further subdivided, but the subtypes tend to differ in coding schemes rather than the demands they place on the network.

Audio data are bidirectional, unstructured (to the delivery service), relatively low bandwidth (8K bytes/second), and real-time. The real-time nature of audio implies that the data flow must be rate and jitter controlled. Audio data are highly sensitive to jitter and loss.

Video data are unidirectional (server to pad), relatively high bandwidth, and real-time. Although InfoPad can accept several format alternatives, the compression scheme currently in use is VQ (compression ratio of about 30:1). Video frames under VQ are 2048 bytes in length, resulting in a data rate of 61.44 kilobytes/second, or almost 500 kilobits/second at 30 frames/sec. This is a significant fraction of the 2Mbps aggregate pad bandwidth, but since only one video stream will be supported per pad sufficient bandwidth remains for the other three data types. Video tolerates a higher proportion of lost frames than audio.

Pen data requires very little bandwidth: about 500 bytes/second from the Gazelle pad currently in use with the hardware link. Pen data are unidirectional and must be delivered with high reliability. The most important performance metric for pen is loop-back latency i.e. the time between pen contact and the appearance of ink on the display. Low latency is critical when working in a draw window, where real pen-like behavior is expected. Since pen recognition will be an important application, pen data must experience no loss.

Text/graphics data are highly bursty and most data units are small. The greatest bandwidth demand occurs when large changes are made in the screen image such as window creation or a screen refresh. Much of the time a text/graphics stream will be composed of cursor information. Text/graphics is non real-time and can tolerate loss.

In summary, the four data types exhibit real-time and non real-time behavior, high and low data rates, a continuum of loss tolerance, and varying jitter requirements. All four data types may be present at the same time in a single pad stream, and multiple pad streams will coexist. The communication channels used for InfoPad will therefore have to exhibit high bandwidth, low latency, reliability, and real-time behavior. In practice, some requirements can be relaxed for some data types if the network interface is parameterized.

## 3 Architecture

This section describes the InfoPad network prototype architecture. The architecture is a general, high level description of what the system should do and how it should look to its users in terms of a programming interface. The mechanisms by which these tasks are accomplished are left to the implementation [7]. The system is intended to be distributed in a nearly arbitrary way across the backbone network; although the software implementation described in this paper imposes no restriction on physical location, integration with the wireless subnet will constrain at least part of the system to reside on machines dedicated to the wireless interfaces. The language of this section is purposefully abstract, since the architecture is a template that can be applied to any compute environment.

The fundamental unit is the *module*. A module performs a specific set of related tasks; the term is intended to underline the modular nature of the InfoPad architecture. Modules communicate with each other through well-defined and tightly controlled interfaces. As an example, microkernel operating systems are based on the premise that functional isolation through modularization is essential for flexibility and power [1] [23]. Major system components such as file systems and virtual memory managers can be implemented as individual modules. An InfoPad module resides entirely on a single compute node, but there can be multiple modules per node.

Modules are combined to form *clusters* and *service groups*. Clusters are differentiated by their view of a pad. One cluster type is permanently associated with a single, specific pad while the other is temporarily associated with a frequently changing set of pads. Connections between members of a cluster tend to be created at start-up and exist for the lifetime of the cluster. A service group is a related collection of modules that provide a specific set of services. For example, one service group generates a "virtual pad" abstraction that hides pad relocation mechanisms i.e. physical location of a pad is irrelevant to a module using this abstraction. The distinction between clusters and service groups will hopefully become clear as this discussion progresses.

Modules are interconnected via generic network services that do not depend on a specific technology. The prototype is *connection-oriented* in the sense that each module maintains state with respect to associated modules: the peer module does not have to be explicitly named on each data transfer at the highest level. The use of the term "connection" in the remainder of the paper means that an association exists. It does not imply that low-level communication protocols are connection-oriented. For example, the connection-oriented nature of the prototype is maintained if either TCP or UDP are used as the network/transport level communication protocol. The terms *connection* and *association* are used interchangeably throughout this paper except in the context of communication protocol implementations, where a different meaning will be indicated.

The architecture also contains two *algorithms* that specify methods for carrying out critical system tasks. Modules, clusters, service groups, interconnects, and algorithms are discussed in more detail in the following paragraphs.

## 3.1 Modules

There are four primary module types: the Gateway, the CellServer, the PadServer, and the TypeServer. There is one secondary module type called the Emulator. The TypeServer is further divided into four subtypes, each dedicated to a particular InfoPad data type. Figure 2 shows one instance of each type and subtype with primary connections.

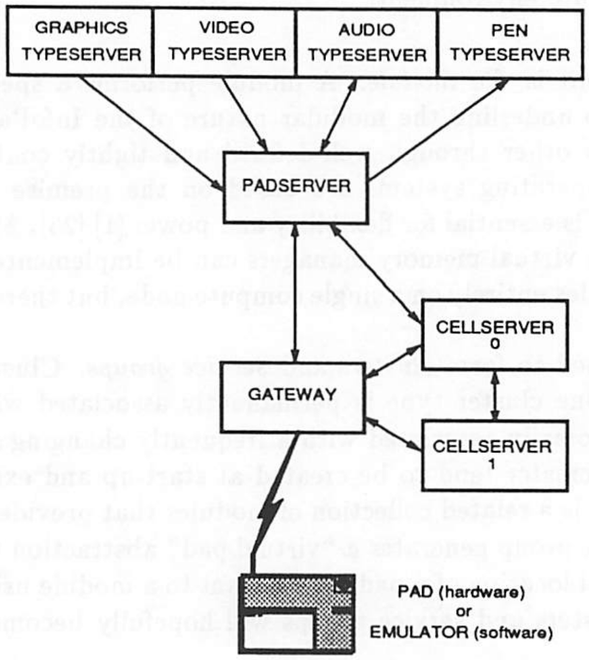

Figure 2: InfoPad network topology: basic view.

Communication with a module takes place through a *port*. A module is connection-oriented between ports; if two ports are associated internally, the module maintains state describing that association.

The architecture specifies the basic function of each module i.e. that necessary to construct a working InfoPad system. Developers are free to add capability provided the basic function of the system is not impaired.

### 3.1.1 Gateway

The Gateway is responsible for switching multiple streams of pad data between the wired subnet and the wireless subnet. The architecture imposes no restriction on the number of pad streams or the nature of connections, although it does maintain a distinction between wired and wireless peers. The Gateway does not operate on data in a stream except to determine and manipulate source, destination, and control information. The Gateway must be capable of switching many pad streams in a fair and efficient manner. The term Gateway arises from the modules role as an interface between two dissimilar networks.

### 3.1.2 CellServer

Each radio cell associated with a Gateway is assigned to a specific CellServer. The CellServer is responsible for traffic management in that cell, including quality of service, error rate detection and control, power management, and participation in connection remapping when a pad relocates. Collectively, CellServers provide a control mechanism for the Gateway; all CellServers associated with a specific Gateway must be fully interconnected, and must be capable of dialog with CellServers associated with neighboring Gateways. Although figure 2 shows only two CellServers, the architecture places no limit on the number of cells under a Gateway. The CellServer has a "session-oriented" view of pad activity, that is, it maintains parameters for a current PadServer-CellServer-Gateway association but does not understand the composition of the data stream.

### 3.1.3 PadServer

The PadServer is responsible for traffic management for a specific pad. Its duties include multiplexing type-specific data into a single type-interleaved stream, scheduling of service on each data type, buffering, and demultiplexing of the pad stream into several typed streams. The PadServer maintains an association with a unique Gateway/CellServer pair as long as its pad resides in the cell under control of the CellServer. When the pad moves into an adjacent cell under the Gateway, the CellServer connection is remapped. If the pad moves to a cell under a different Gateway, both the CellServer and Gateway connections are remapped.

### 3.1.4 TypeServers

The TypeServer is the management agent for a particular data type for a specific pad. For instance, the video TypeServer is responsible for assembling video traffic from all sources into a single stream that is then passed to the PadServer. Generally, the video TypeServer will choose a single video source to play at any given time.

There are four TypeServer subtypes, corresponding to the four InfoPad data types: audio, video, pen, and text/graphics. The PadServer allows only one of each subtype to be attached at any given time.[2] It is anticipated that TypeServers will define the application programming

[2]This restriction may be relaxed in the future to accommodate new video transmission techniques.

interface to the InfoPad network, although TypeServers can themselves be viewed as an application if the programmer is willing to provide necessary PadServer interface code. In other words, the TypeServer is intended to present an abstraction of the pad that is defined by the TypeServer author. In reality, TypeServers and applications will probably be developed in tandem until a better understanding of InfoPad traffic characteristics is developed.

The main reason for the division of responsibility between PadServer and TypeServers is to relieve the PadServer of application-specific tasks in order to limit PadServer function to a manageable subset and allow PadServer implementations to stabilize quickly. Since a number of developers will be working independently on type-specific aspects of the project, a well defined and consistent programming environment is essential (see section 3.4.3).

One criticism of this organization is that TypeServers add an additional level of latency-inducing overhead. Measurements show that this is not a problem for the current implementation [8].

### 3.1.5 Emulator

Eventually, the InfoPad world will be equipped with a mature infrastructure including network systems, wireless links, and hardware pads. Unfortunately, it will be some time before the project can offer these facilities to software designers. To bridge the gap and allow development to proceed, a software entity called the *Emulator* (or *Emu* for short) has been defined that emulates the actions of a hardware pad. As much as possible, Emu has been designed to look like a pad when connected to a Gateway. Emu also provides a platform for simulation of pad capabilities. The Emulator is not a part of the architecture per se, but for the sake of clarity it is referred to as a secondary module.

In a canonical InfoPad system, the Gateway must reside on a compute node connected to wireless hardware while the CellServers can be anywhere on the network. In the software prototype implementation, the Emulator removes this constraint. Emu also allows the system to reside entirely in a relatively small, self-contained release directory. It is easy, therefore, to bundle the system and move it to a different network. Completely independent systems can be created to support independent research.

## 3.2 Clusters

As described above, a cluster is a functional grouping of modules that differ primarily in their relationship to a specific pad. Associations between modules in a cluster tend to be stable i.e. once established, they are not likely to change. There are two types of clusters: the *pad cluster* and the *gate cluster*. Figure 3 is a copy of figure 2 with clusters identified. The InfoPad naming scheme, described in section 3.4.2, is based on the cluster. Each cluster is assigned a global number and each module within a cluster is assigned an identifier unique to that cluster. The module identifier is composed of a type specifier (e.g. Gateway or Padserver) and a module number. This scheme forms a two-level hierarchy.

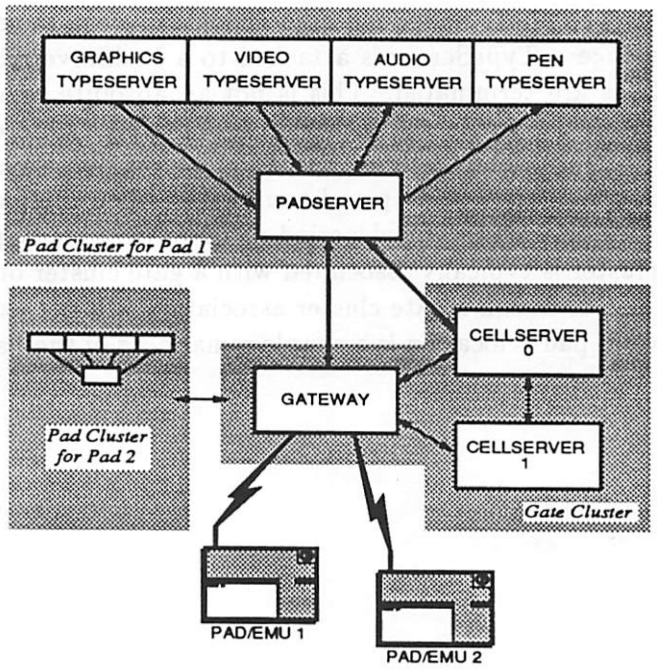

Figure 3: InfoPad network topology: clusters.

### 3.2.1 Gate Cluster

A gate cluster contains one Gateway and all CellServers associated with that Gateway. Gate clusters ideally live "forever" i.e. since the Gateway models a piece of hardware that will eventually be part of the physical infrastructure and CellServers manage radio cells defined by hardware, gate clusters are expected to be long lived in the absence of hardware failure. Network connections between gate cluster members are permanent. Once established, they are never dynamically remapped. The external view of a gate cluster is a set of pad links on the wireless side and a set of pad cluster connections on the wired side.

### 3.2.2 Pad Cluster

A pad cluster is a collection of modules dedicated to a specific pad. It contains one PadServer and (potentially) one TypeServer for each data type. For minimal operation, a pad cluster must contain the PadServer, a text/graphics TypeServer, and a pen TypeServer. A pad user may specify a startup environment, in which case there may be an audio TypeServer, a video TypeServer, and/or applications in the basic environment. The modules in a pad cluster can reside individually on any compute node, and associations between them are generally permanent in the sense that once a TypeServer is attached to a PadServer the association remains in place until all modules are terminated. This is not an absolute requirement, however. A user may choose to replace a TypeServer with another offering a different set of capabilities, but this is expected to be rare. The lifetime of a pad cluster is determined by the pad user; it can be started when the pad is powered up and terminated when the pad is powered down, or it can remain in place for an undetermined period of time (over many power cycles). At any given moment, a pad cluster is typically associated with a gate cluster on one side and a set of applications on the other, although a gate cluster association will not exist if the pad is off or out of range. The affect of pad relocation is a simple remapping of the pad cluster/gate cluster association.

## 3.3 Service Groups

A *service group*, like a cluster, is a grouping of modules for a specific purpose. While clusters are responsible for management of a single pad or a localized set of pads, service groups are concerned with providing system-wide *services* that apply equally to all pads. Also, clusters are static in terms of inter-module connections while service groups are dynamic or have no direct association between members at all. By analogy (in ISO terminology), the network layer of a communication protocol offers a set of services to the transport layer above and makes use of services offered by the data link layer below. The service groups in InfoPad tend to be hierarchically related but are not necessarily so. There are three service groups defined in the InfoPad architecture: *delivery support*, *type support*, and *applications*. Figure 4 is a version of figure 2 with the service groups identified.

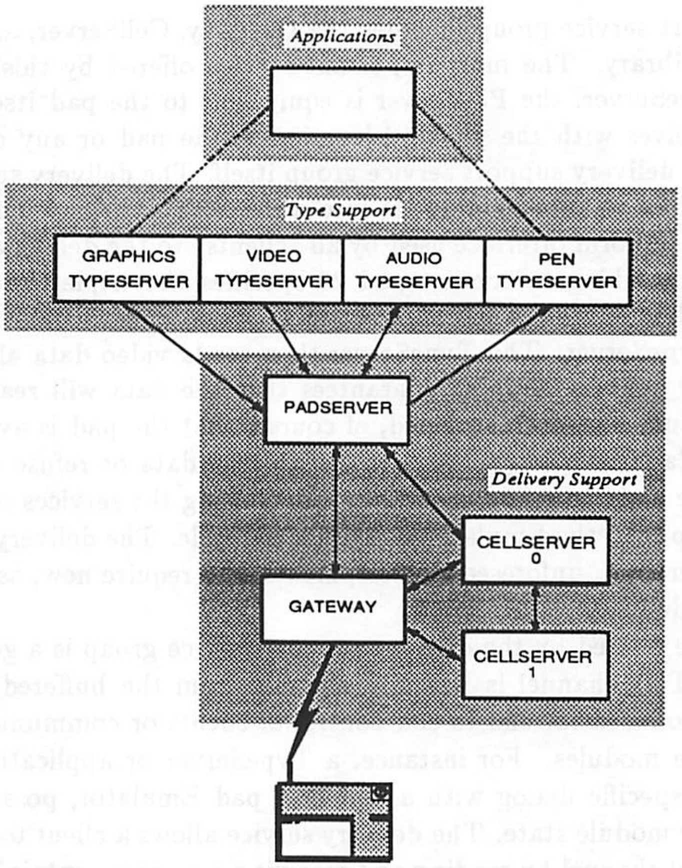

Figure 4: InfoPad network topology: service groups.

One of the essential distinctions between service groups is the level of abstraction they give to a pad. The delivery support service group maintains detailed information about a pad, including physical location, model, radio power level, and current bandwidth consumption. The type support service group understands the concept of a pad but none of the specifics. It views the pad as an entity at the other end of a network connection that obeys a set of rules imposed by a well understood interface specification. The application service group can be completely abstracted from the pad. For instance, an application can be a window system client conducting a dialog with a text/graphics TypeServer acting as a window system server. Figure 4 shows the loose hierarchical relationship between service groups.

While the cluster is an important concept, the prototype implementation is organized around service groups because of natural programming interfaces created by pad abstractions. A programmer working on one service group has little need for the implementation details of another service group - the requirements for interconnecting service groups can be made very simple.

### 3.3.1 Delivery Support

The delivery support service group includes the Gateway, CellServer, and PadServer modules and a distributed library. The most important service offered by this group is the "virtual pad" i.e. to a TypeServer, the PadServer is equivalent to the pad itself. TypeServers need not concern themselves with the physical location of the pad or any other entity, including the members of the delivery support service group itself. The delivery support library includes a name database that maps a module ID into a network location. This database is hidden beneath a standard uniform interface used by all "clients" to the delivery service. For example, a video TypeServer wishing to contact pad 25 specifies the triple (PadServer, 25, video) to the interface, and the delivery support library locates and makes connection to the PadServer on behalf of the TypeServer. The TypeServer then sends video data along the connection to the PadServer, and delivery support guarantees that the data will reach the pad regardless of its current or future location (provided, of course, that the pad is available). If the pad is not available, the PadServer may choose to discard the data or refuse connection. Although TypeServers are the most common client, any entity using the services of delivery support is a client of delivery support; the Emulator is another example. The delivery service is designed to be flexible and general i.e. unforeseen developments may require new, as yet undefined, clients to make use of the delivery service.

A second service offered by the delivery support service group is a generalized out-of-band control channel.[3] This channel is logically separate from the buffered data channel, and is designed to be customized for end-to-end control of clients or communication between clients and delivery service modules. For instance, a TypeServer or application is likely to engage in type or module specific dialog with a pad or a pad Emulator, possibly to set or retrieve parameter values or module state. The delivery service allows a client to carry on an arbitrary dialog on the control channel by sending and receiving messages containing formatted data [1]. Delivery support does not interpret the contents of the payload unless instructed otherwise e.g. when a delivery service module is the destination. The CellServer is expected to be the most common endpoint for a delivery support module/client (internal) dialog.

[3]In-band implies that data sent first will arrive first i.e. an ordering is enforced. An out-of-band channel can circumvent this ordering; a data unit sent out-of-band can arrive at its destination before data units that were sent earlier. Out-of-band channels are typically used for priority data.

Any services offered by the member modules themselves such as bandwidth control, multiplexing/demultiplexing, and error detection/correction are also part of the delivery support service set.

The delivery support service group is a user of its own services. That is, each delivery support module uses a variant of the general interface for inter-module connection, and a control channel exists between each connected pair of delivery service modules. The set of control events available to delivery service modules is somewhat richer than the set available to clients.

### 3.3.2 Type Support

The type support service group encompasses TypeServers and any additional modules required to provide a type based interface to an application. Type support modules may consult each other for cross-type information exchange. For instance, a pen TypeServer will notify a text/graphics TypeServer when the pen changes position so that a cursor can be moved or a window exposed. An audio TypeServer may perform data conversion for a voice recognition application. In the first case, the text/graphics TypeServer is making use of the service offered by the pen TypeServer. In the second case, the voice recognition application is making use of services offered by the audio TypeServer. In turn, the type support service group makes use of the services offered by the delivery support service group. Sometimes the distinction between applications and type support modules is fuzzy: voice recognition may ultimately be the role of the audio TypeServer.

Although division of type support module function is well defined in terms of data type represented, the services offered are not. These will depend on demands made by applications not yet written, and in some cases not yet visualized. Even though delivery support is **internally** subject to change, its service set is stable in comparison to type support. This underscores the need for a strong functional division between the type support service group and the delivery support service group. Additionally, the roles of the two service groups are very different. A "firewall" interface is essential.

### 3.3.3 Applications

This service group propagates InfoPad services to users. It is the least defined of the three groups. The architecture does not specify the structure of the application service group; it is identified here to complete the pad-to-user picture. There is currently only one application built specifically for InfoPad: a window program that presents a view of a note pad. The main use of the Notebook application has been as a sink of pen data and source of text/graphics data as part of an effort to measure pen latency [8].

## 3.4 Inter-Module Communication

InfoPad Inter-Module Communication (IMC) is based on a connection-oriented packet based model. Connections are maintained at a high level; underlying communication protocols can be reliable or unreliable,[4] connection-oriented or connectionless, stream or datagram, lightweight or heavyweight. Physical media can be point-to-point, bus, ring, or other. Elements central to IMC are ports, IDs, interfaces, and packets. Each of these elements will be discussed below.

IMC exists as a part of each module and in the delivery support service group library. The library contains algorithms for address mapping, packet manipulation, and connection establishment using a specific protocol. Several versions of the library exist, each using a different low-level communication protocol. To support the goal of flexibility, most implementation dependent function is part of the library so that, if a different protocol implementation is needed, much of the work needed for the change is performed by simply using a different library. However, each module must include some non-general i.e. protocol specific IMC support. Ports, in particular, are created and manipulated by the modules themselves. They are included in the discussion of IMC because they are the IMC end points. All modules that communicate with other modules must incorporate the library (it's hard to conceive of a module that won't need communication).

### 3.4.1 Ports

Within a module, the point of interaction with other modules is the port. There are several varieties of ports e.g. the *link port* is used by a Gateway for connection to a pad while the *PadServer port* is used for connection to a PadServer. Each PadServer has up to four *TypeServer ports.* Every module has a *monitor* port, which is dedicated to listening for a new connection and accepting the connection if satisfied with the results of a brief handshake.[5] In general, a separate connect point will be spawned for each connection accepted by the monitor port. For portability (flexibility) reasons, there are no hard coded addresses. All ports are created and initialized through contact with the monitor port; this action is transparent to the calling module. Ports differ mainly in semantics and their view of module data buffers.

### 3.4.2 IDs and Address Mapping

InfoPad network modules are uniquely identified by the triple *(module type, cluster number, module number).* This triple is referred to as the module identifier, or ID. For Gateways and PadServers, module number is synonymous with cluster number; for a CellServer, module number is an identifier specifying a cell; for TypeServers, the module number is an identifier specifying data type. In the prototype implementation described in section 4 there is only one

[4]If the protocol is unreliable, IMC must implement error-free transmission algorithms for control information and data types that require low or zero error transmission, such as pen. Otherwise, IMC uses the services of standard reliable communication protocols such as TCP.

[5]Similarity with Unix SOCK_STREAM terminology is not coincidental; the most common port implementation uses TCP. However, even when other protocols have been used (e.g. UDP) the method of "listen/accept" is still employed, although with significantly different semantics.

CellServer per Gateway, so module number is omitted for the CellServer in that case. The system architecture envisions a globally unique ID for each pad similar to an Internet address.[6] This global ID is the cluster number for pad cluster modules. Example ID triples for each primary module are shown below. The triples with duplicated arguments can be reduced to pairs in certain contexts.

- Gateway: ("gw", Gateway #, Gateway #)
- CellServer: ("cs", Gateway #, cell #)
- PadServer: ("ps", pad ID, pad ID)
- TypeServer: ("ts", pad ID, data type)

The IMC portion of the delivery support service group library implements mapping of these triples to physical network locations, including all technology-specific tasks. The library initiates a connection with the help of delivery support modules involved in the connection, terminates a connection, and defines the semantics of all control messages.

The sequence of events for address mapping goes something like this:

1. A module requests connection with another module by making a call on the library, specifying a triple.
2. A library procedure consults a name database in which all currently active members of the delivery service have registered themselves using another library procedure. From the database, an entry is retrieved that specifies a technology specific mapping for the monitor port. For instance, if the communication protocol is TCP, the mapping will be (module type, cluster number, module number) to (Internet address, TCP port number).
3. Connection is attempted to that network address.
4. If connection is successful, the requesting module is notified that data transfer can begin. If the connection is unsuccessful, the requesting module receives an error message and the connection is denied.

### 3.4.3 Interfaces

An interface is a set of procedures and specifications that define a clean and consistent programming environment to module designers. The interface hides implementation specific details from the programmer, so that code using the interface is portable. This is a central aspect of the firewall interface between delivery support and type support service groups.

IMC interfaces come in two flavors: external and internal [18]. External interfaces define the boundaries of the delivery support service group (the firewall), and are used by TypeServers

[6]In fact, pad ID's can be actual Internet addresses in future implementations. The current prototype is limited to 256 pads by restrictions in the packet header address space.

and Emulators. Internal interfaces are structurally similar to external interfaces, but they are subject to modification. Internal interfaces are used by delivery support modules to conduct internal business. Interfaces between delivery service clients (e.g. TypeServer to TypeServer) and between clients and applications are not part of IMC.

Each interface offers two channels: data and control. In basic form, both interfaces support four procedures on the data channel that employ familiar open/read/write/close semantics:

**connect** (port, destination, arglist)
Connect *port* to *destination* with parameters *arglist.*

**read** (buffer, length, port)
Read *length* bytes into *buffer* from *port.*

**write** (buffer, length, port)
Write *length* bytes from *buffer* to *port.*

**close** (port)
Close *port* and free resources.

These procedures can be configured to operate on a byte stream (packet headers removed) or on packets. That is, the data returned from read() is either payload only or full packets while data sent with write() can be payload only (the interface adds the header) or complete packets. Because the architecture is designed to work with byte stream communication protocols like TCP, there are no explicit message boundaries defined by a read() invocation: message boundaries (which are only meaningful when reading packets) are indicated by the value in the length field of the packet header.[7]

Both interfaces define two basic procedures on the control channel:

**get** (buffer, length, port, moduleID)
Get *length* bytes from *moduleID* over the control channel for *port* and put in *buffer.* The module specified by *module ID* must be accessible (or equivalent) to the module associated with *port.*

**set** (buffer, length, port, module ID)
Send *length* bytes from *buffer* on the control channel for *port* to *moduleID.*

Get() and set() are similar to read() and write(), except that get() and set() operate on the out-of-band control channel and can (in theory) send a packet to an arbitrary module. In practice, source/destination pairs will probably be limited to a subset of modules (e.g. TypeServer to Emulator but not Emulator to Emulator). The extent of connectivity depends on the sophistication of module implementations.

An internal interface also defines procedures in the following classes:

[7]In message passing systems, read and write operations transfer structured data units larger than a byte. Message boundaries are implied by the operation i.e. one message is passed per call. In this architecture, packets are the analog of messages.

**disconnect** (port)
: Disconnect *port*. Used only with connectionless low-level protocols.

**fwdreq** (source ID, destination ID, arglist)
: Forward a request for connection from module *source ID* to module *destination ID* with parameters *arglist*.

**terminate** (cluster #)
: Kill all modules in *cluster #*. Cluster # is a Gateway or PadServer number.

**probe** (port)
: Test a module to see if it's alive. Intended for use in situations where a module is out of resources and wants to make sure all associated modules really need what they're using. Similar to Unix *ping*.

**source quench** (port)
: Stop a data stream originating from peer module at *port*.

**source resume** (port)
: Start a data stream originating from peer module at *port* that has been stopped.

Each of these has a corresponding acknowledgement procedure and a corresponding response processing procedure. Fwdreq() is used exclusively to forward a pad connection request from a Gateway to a CellServer. A function can be provided that will allow users to initiate a terminate request.

### 3.4.4 Packet Format

InfoPad packets create a level of abstraction over low-level technology, allowing the prototype to be used as a testbed for InfoPad research; it can be run over any network, and packet size and format can be manipulated at will. Packets also enable multiplexing of virtual InfoPad connections over a single network address association. Furthermore, packets allow relatively fine-grain resolution of a data stream to aid in priority scheduling and to enhance the appearance of concurrency. The packet format is shown in figure 5.

The *sync* field is an artifact of an early version of pad hardware and is no longer necessary. However, it has proven to be such a valuable debugging tool that it has been retained in the current prototype despite the added overhead. It allows the system to self-recover in the presence of all too common coding bugs. The *core* field contains InfoPad link-level information. The triples *(module type, cluster #, module #)* are source and destination module IDs, as described previously. The *info* field holds context specific information. *Code* refers to the packet type i.e. data or one of several control types. *Seq* indicates the position of the packet in a control sequence (see section 3.5.2). The remaining 24 bits of the info field are formatted in various ways. For example, a data packet carries payload type and length, while the info field of a connection acknowledgement control packet carries a result code. Some common fields such as CRC are omitted, since these "user-level" packets are expected to be encapsulated in low-level protocol data units over which CRCs are computed.

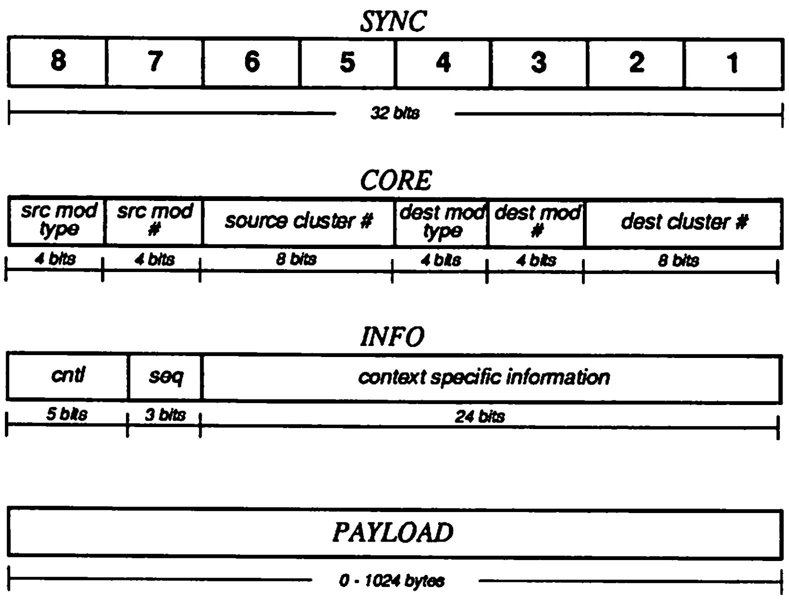

Figure 5: InfoPad packet format

## 3.5 Algorithms

The definition of algorithms is generally left up to an implementation. However, this architecture defines two critically important algorithms. One is related to pad mobility, and the other is concerned with maintaining parallelism in data streams when servicing a control request and ensuring reliability of control messages regardless of the low-level communication protocol in use.

### 3.5.1 Pad Relocation (handoff)

One of the most critical operations in an InfoPad system is the transfer of a pad to a different cell, called *relocation* or *handoff*. This is a global action; several modules must cooperate, so the algorithm cannot be defined for a specific module. The pseudocode below is an approximation of the relocation algorithm that does not take into account multiple CellServers per Gateway. It is presented as an example.

The specifications described here are skeletons that provide simple and easily implemented relocation functions for the prototype. It is anticipated that these algorithms will be enhanced as the sophistication of the system grows.

There are two basic cases and two basic conditions that must be considered:

- **Case I**: Pad enters cell. It could be a pad that has just been switched on, a pad moving into a new cell, or a pad re-entering a cell before its state has been removed from the cell.
- **Case II**: Pad exits cell. It could be a pad turning off, or a pad moving to another cell.

- **Condition 1:** Pad cluster for the pad exists.
- **Condition 2:** Connection from the pad cluster for the pad to any gate cluster exists.

There are three possibilities for the two conditions (pad cluster doesn't exist/connection exists is impossible). Each of the three must be handled for each Case. In order for the algorithms to be simple, four assumptions must be made:

1) Gateways and CellServers always exist and are in fixed locations once started.

2) PadServers and TypeServers may or may not exist, and if they do exist can be anywhere on the local net.

3) Each pad has a heartbeat and can send a control packet containing its global pad ID to a Gateway.

4) Modules are trusted i.e. no authentication is needed.

These assumptions are reasonable within the context of the architecture. Pad hardware under development will be capable of realizing assumption 3. If assumption 4 becomes a problem, enhancements to the algorithms may be necessary. At present, authentication is not implemented.

Part of this algorithm is implemented in the PadServer and part in the CellServer. In each module, the algorithm is divided into two sections, one for Case I and the other for Case II.

## PadServer

### Case I

```
if (PadServer is connected to correct gate cluster already) {
    nothing happens
}
else if (PadServer exists [condition 1]) {
    accept connection from new CellServer
    if (connection exists to another gate cluster [condition 2]) {
        remove connection to old CellServer
        remove connection to old Gateway
    }
    receive Gateway ID/pad ID from new CellServer
    check name database for location of Gateway
    establish connection to new Gateway
}
else (PadServer doesn't exist) {
    get started by CellServer
    register self in name database
    accept connection from CellServer
    receive Gateway ID/pad ID from CellServer
    check name database for location of Gateway
    establish connection to Gateway
    start TypeServers
    establish connection with TypeServers
}
```

### Case II

PadServer takes no explicit action.
Connections will be re-mapped if the pad enters another cell.
See CellServer Case I.

## CellServer

### Case I

```
pad sends pad ID to CellServer via Gateway
if (correct pad state exists - maybe pad left cell momentarily) {
    do nothing
}
else if (maximum pads in cell already) {
    deny connection
}
else {
    check name database for location of PadServer
    if (entry for PadServer exists [condition 1]) {
        attempt connection to PadServer
        if (connection fails - maybe PadServer crashed) {
            start a PadServer on some machine
            establish connection to new PadServer
        }
    }
    else (PadServer doesn't exist) {
        start a PadServer on some machine
        establish connection to new PadServer
    }
    send Gateway ID and pad ID to new PadServer
}
```

### Case II

```
if (timeout) {
   terminate connection with Gateway
   terminate connection with CellServer
   optionally terminate the pad cluster
{
else {
   no action
{
```

### 3.5.2 Control

All control information sent by a module must be received by the destination, and it must be received without error. Since the degree of reliability of the underlying communication protocols is not defined by the architecture, an algorithm is included that will ensure reliable transfer of control packets. This algorithm is based on a technique used successfully by Cheriton and others, [11] [5] [19] and is called *little brother* (InfoPad term) due to its habit of hanging around until it gets what it wants.

The control algorithm operates on a request-reply basis. A module wishing to initiate some action with a peer transmits a control packet from a port. The receiving module performs the action (or doesn't) and returns an ack (or nak) to the initiating module. The algorithm manages both sides of this dialog. There is an instantiation of the algorithm in every module.

The algorithm makes the pessimistic assumption that a packet will be lost. When a control packet is sent, it starts a timer and returns control to the caller. When the timer expires, another packet is sent if an acknowledgement from the first has not been received. This process continues for a pre-defined small number of tries before the algorithm gives up and reports an error. The system is not allowed to begin another control action of the same type (e.g. connect request, set request, etc.) until the current action has been acknowledged.[8] Because control packets are relatively infrequent in this architecture and the number of retransmissions is small, there is no danger of swamping the network with control packets provided the implementation is correct.

In an unreliable network, packets that are otherwise undamaged can be lost, delayed, or misordered. Since a number of identical control packets can be outstanding at one time (the first try and any retransmissions), it is possible to receive an acknowledgment for a previous control action after a new, subsequent, control action of the same type has been initiated. In this case, the first action has been acknowledged already and the second is waiting for an ack. The errant acknowledgement will then cause the second action to believe it is complete if additional measures are not taken.

To prevent this occurance, each control packet is assigned a sequence number (the *seq* subfield of the info header field). The first packet in a control action and any retransmissions get the same value. When an ack arrives for a particular value, the sequence number is advanced for that control type only. If an ack arrives with a number less than the current sequence value, it is discarded. A sequence is defined for each control packet type. The seq subfield is three bits in length; it is expected that by the time the sequence number wraps around, all previous control packets and acks with the next value have been permanently lost or discarded. The retransmission timers can be set to ensure that this is true.

Additionally, the "control space" for each port is orthogonal with respect to all other ports, since a port is uniquely defined by the core field of the packet header.

One question that may be asked is: "If pen data must be reliable, why not use whatever reliable low-level protocol is on the pen data channel for everything?" The answer is in keeping with the attempt to limit functionality where possible. Pen data channels are one of several

[8]A control action encompasses the time between transmission of the first packet for a specific control type and the acknowledgement of that first packet or a retransmitted packet.

channel types. Others, like text/graphics channels, do not require the same level of reliability. Data and control channels often coexist on a single network link; if the link does not require high reliability for data, the system can use a simple unreliable transport/network layer protocol. The little brother algorithm adds the required reliability for control information while retaining low overhead.

When a control action is pending, other actions on the port may be delayed or disabled. The algorithm uses a table to determine which actions are allowed at any given time. Table 1 shows an example for eleven control types.

Table 1

| Control Type | Recv Data | Send Data | Recv Cntl | Send Cntl | Comments |
|---|---|---|---|---|---|
| connect | yes | no | yes | no | data pkt = ack |
| fwdreq | yes | no | yes | no | data pkt != ack |
| disconnect | yes | no | yes | no | data pkt != ack |
| terminate | – | no | yes | no | wait for ack |
| source quench | yes | yes | yes | yes | advisory only |
| source resume | yes | yes | yes | yes | new pkt is ack |
| get | yes | yes | yes | yes | |
| set | maybe | yes | yes | yes | |
| probe | yes | no | yes | no | new pkt is ack |
| ack | – | – | – | – | never pending |
| nak | – | – | – | – | never pending |

A copy of the table is maintained by each port (there may be small differences in content). Each port also keeps state for each control type (e.g. current sequence number or action pending). On arrival of a control packet or expiration of a timer, the table and state are consulted to determine which action to take, if any.

For several control types, a new data or control packet serves as an acknowledgment. Source quench is advisory only: the receiving module is not required to take action. Positive and negative acknowledgments are sent but not monitored. If an acknowledgment is not received, another request for action will eventually be issued. For this reason, control actions must be idempotent.

## 4 IPN: Implementation of the Delivery Support Service Group

This section presents an implementation of the delivery support service group. This paper will cover delivery support implementation in greater detail than type support for two reasons: because the development of delivery support in the prototype is somewhat more advanced (delivery support had to be operational before work on type support and application modules could proceed in earnest), and the delivery support service group is a brand new, fully custom implementation while type support relies heavily on well documented software. This implementation of delivery support is called InfoPadNetwork, or IPN.

The general approach and methodology in the development of IPN stresses efficiency of data transfer. In particular, great care has been taken to limit operating system interaction; IPN minimizes the use of system calls and does all critical path processing in a single user context. When system calls are necessary, they perform as much work as possible to maximize the time between invocations. There has been significant recent work in support of user-level execution of traditional kernel tasks to avoid the overhead of invoking the kernel [2] [3] [4].

Another measure employed by IPN to enhance efficiency and fairness is the use of "pseudo" concurrency in port service algorithms. In this approach, all operations are non-blocking, and no port is allowed to read more than one packet or write more than one buffer at a time.[9] Therefore, ports will not starve and the wait time between service of a particular port is bounded by the time required to service all other active ports and ready buffers once (in the absence of an involuntary context switch).

This section will begin with a discussion of functions and features common to all delivery support modules. Each module will then be described in turn. The section will conclude with a description of the IMC library.

### 4.1 Physical Environment

The prototype is currently in operation on a general-purpose Ethernet in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley. This network provides communication for about sixty hosts, and is used on a daily basis for both research and administration. Most network compute nodes are Sun Sparc workstations, primarily Sparcstation 2's and 10's running SunOS 4.1.3. All standard Sun networking facilities are supported including an implementation of the Internet protocol suite (TCP/IP), NFS, and RPC,

InfoPad modules are implemented with Unix processes, and kernel interaction is through common system calls. No kernel modifications were required except in one case related to the hardware link [8]. The programming language used is ANSI 'C'.

Most network connections use the services of TCP. The discussions that follow assume TCP connections. Implementations using UDP and NIT have been created, installed, and tested -

[9]In traditional request-reply semantics such as RPC, the requester "blocks", or waits, on a system call until the requested action is possible. There is often no guarantee that the request will be serviced. This means that the requester could wait indefinitely.

use of any protocol other than TCP will be noted where appropriate. See [9] for a detailed discussion of these implementations.

It is expected that the prototype will quickly outgrow Ethernet. Work in progress as of this writing include a port to Sun Solaris and installation of a limited ATM network specifically for InfoPad.

## 4.2 Common Functions and Features

IPN modules are built from a common framework. Differences arise primarily in the organization of internal buffers and data paths, but several important components are virtually identical in all module types.

### 4.2.1 Ports, Procedures, and State

As described in section 3.4.1, ports are the communication endpoints for modules. Associated with each port is a read procedure, a write procedure, and a state structure. Read and write procedures implement port service algorithms, and the state structure contains information related to the identity and current state of the port. The read and write procedures discussed here are distinct from the read and write procedures discussed earlier in the context of the interface. Port procedures are specific to IPN modules while interface procedures are public access points to IPN.

Modules internally associate each port with at least one other port in the same module. For instance, the Gateway associates each PadServer port with a unique pad link port. The PadServer associates its single Gateway port with all TypeServer ports. A packet read from a port will be placed in a buffer that is attached to the output of the associated port, and vice versa. In other words, a read procedure will transfer a packet from a port to a buffer and a write procedure will transfer the contents of a buffer to a port. Neither procedure has explicit knowledge of the associated port. Routing information is implied in the port-buffer mappings.

When a port is created, a state structure instance is allocated and initialized with items such as port name (related to the connecting module ID), pointers to read and write procedures, a custom I/O package, pre-assembled packet headers (more on this later), connection endpoint protocol addresses, and several implementation specific parameters. Gateway and PadServer port structures also contain reference pointers to internal data buffers.

IPN port algorithms are implemented as finite state machines. This approach allows parallelism at the packet level while maintaining a simple structure. Originally, lightweight processes were considered for implementation of the state machines, but since the machines are all relatively simple (another concession to efficiency), the overhead and complexity of LWPs is unacceptable given the limited enhancements they offer in comparison to the switch based approach finally adopted.

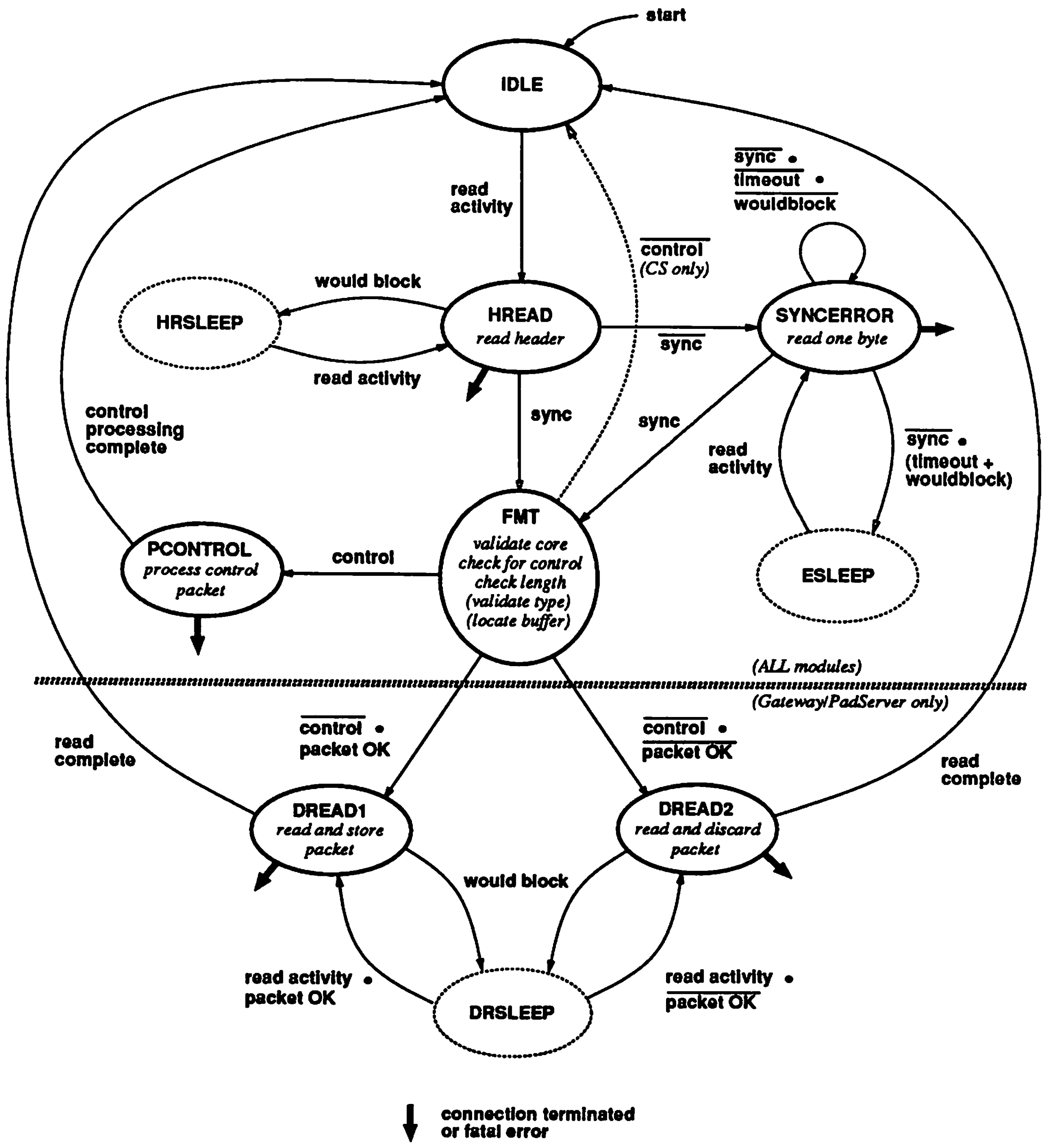
start
IDLE
read activity
control (CS only)
sync • timeout • wouldblock
would block
HRSLEEP
HREAD read header
SYNCERROR read one byte
read activity
sync
sync
sync
control processing complete
read activity
sync • (timeout + wouldblock)
PCONTROL process control packet
control
FMT validate core check for control check length (validate type) (locate buffer)
ESLEEP
(ALL modules)
(Gateway/PadServer only)
read complete
control • packet OK
control • packet OK
read complete
DREAD1 read and store packet
DREAD2 read and discard packet
would block
read activity • packet OK
read activity • packet OK
DRSLEEP
connection terminated or fatal error

Figure 6: Module read procedure generic finite state machine.

There are two generic state machines and associated procedures. Not surprisingly, one is dedicated to reading and the other for writing. Figures 6 and 7 show the read and write state machines, respectively. Differences between port algorithms are concentrated in the *format* state (FMT) and below e.g. the CellServer does not include *data read* (DREAD) or *data read sleep* (RDSLEEP) states because it expects to receive control packets only. The FMT state in figure 6 lists typical actions; those in parentheses apply to the Gateway and PadServer only.

A read procedure is invoked by the *scheduler* when activity is detected on a port. A write procedure is invoked by the scheduler when the contents of a buffer assigned to the port exceed a pre-defined threshold. The procedure call includes a pointer to the port state structure and, possibly, a pointer to a buffer (some state structs have pointers to all relevant buffers, so an explicit pointer argument is not needed). The procedure first enters a switch that tests a state field to determine if the port is *idle* or *sleeping*. If it is idle (the common condition), the state machine is entered at idle. If it is sleeping, the state machine is restarted at the appropriate sleep state. Sleeping means that the procedure was unable to complete a full cycle of the state machine in its last invocation.

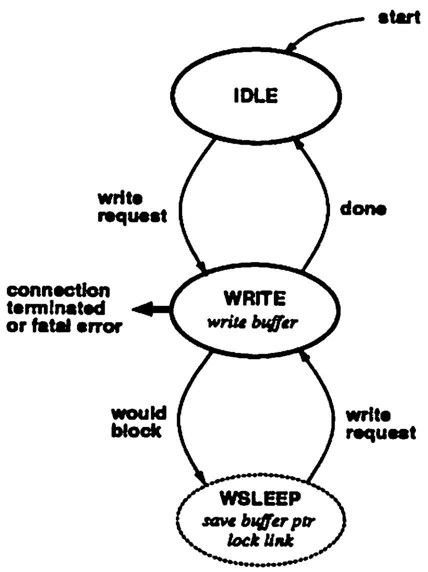

Figure 7: Module write procedure generic finite state machine.

The read procedure will attempt to read a single packet on each invocation. If it is unsuccessful for some reason, it marks the port as sleeping on a read and returns to the scheduler. The write procedure will attempt to write an entire buffer to a port (since a port write ultimately involves a system call, the procedure tries to write as much as possible at one time). Like the read procedure, if the write is unsuccessful (ALL of the buffer must be written) it marks the port as sleeping on a write and returns. The "buffer write" policy is a write-side manifestation of the attempt to limit system calls. The read-side equivalent is the custom I/O package discussed in section 4.6.1.

If certain errors are detected on the port (for instance, the network peer has closed its end of the connection) a read or write procedure will mark the port for closure and return. When the scheduler has finished servicing each port it reviews a global table for closed ports. If any are found, a close procedure is called to dispose of port state and dismantle any internal structure associated with that port. This *delayed close* eliminates a problem associated with dangling references caused by closed ports still on the service list (see section 4.2.2).

### 4.2.2 The Scheduler

The scheduler is implemented by a set of procedures that are responsible for detecting read activity on ports, testing buffers for the "threshold" state, and scheduling port service. If no ports or buffers require service, the scheduler sleeps on the *select()* system call. When read activity is detected, the scheduler invokes a procedure called *Service*(). Service() determines which ports require service and invokes the read procedure for each port according to a priority algorithm. The scheduler code is structured so that the priority algorithm is easy to replace without disturbing the rest of the scheduler. This way, experimentation can be done with different methods of scheduling. The default method is first-come-first-served.

After each active port has been serviced, Service() checks all module buffers and schedules a write for each buffer in which the threshold has been exceeded, again according to the priority algorithm. Once the buffers have been written, Service() scans a port table for closed ports, calling a close procedure for each. The scheduler then returns to the select() function. This time, select() does not sleep in the absence of read activity; it performs a quick poll and returns to the scheduler regardless of what it found. If there is new activity, the sequence described above is repeated. If not, the scheduler calls *IdleService*() to flush all buffers, then goes to sleep on select().

The scheduler maintains a table called the *service list* that holds information about all ports ready for service (see figure 8). The service list is an array of pointers to structures, each structure containing an operation flag, a port structure pointer, a buffer pointer, and a pointer to a service procedure. Normally, the operation flag is set to OP_READ or OP_WRITE. However, if there is read activity on the monitor port, a peer module is probably requesting a new connection. In this case, the operation code is set to OP_CONN and the monitor port read function, when invoked, accepts a new network connection and initializes a new port. The connection is not immediately validated; the port exists as an *orphan* until the next scheduler cycle occurs, at which point a connection request packet is expected on the new port. If the request is approved the port is *adopted* and becomes a member of the active port set. Otherwise, the port is closed.

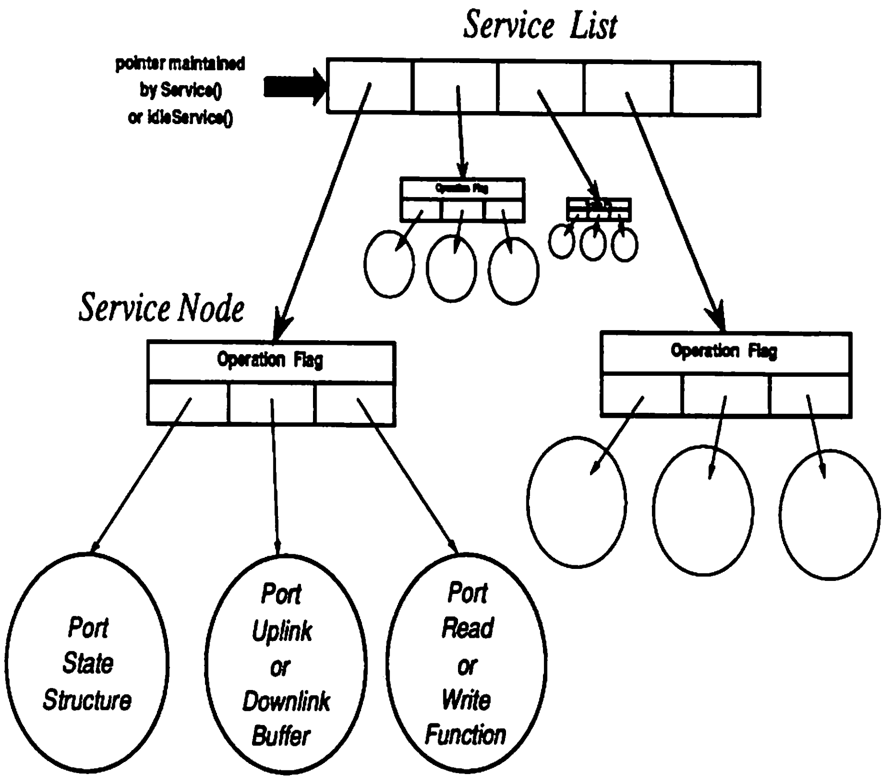
Service List
pointer maintained by Service() or idleService()
Service Node
Operation Flag
Operation Flag
Port State Structure
Port Uplink or Downlink Buffer
Port Read or Write Function

Figure 8: Scheduler service list structures

Simplified pseudocode for the scheduler is shown on the following two pages. This pseudocode is very general, and does not take into consideration many small but important implementation details.

```
while (FOREVER) {
    sleep or poll on select()
    if (read activity detected) {
        call Service()
    }
    else {
        call IdleService()
    }
}
```

```
Service()
{
    for all ports {
        if (monitor port is active for read) {
            add monitor port struct to service list
        }
        if (any other port is active for read {
            add port struct to service list
        }
    }

    call priority procedure with service list

    for all buffers {
        if (buffer contents greater than threshold) {
            add port struct and buffer pointer to service list
        }
    }

    call priority procedure with service list

    for all ports {
        if (port marked for closure) {
            call close function on port
        }
    }
}
```

```
IdleService()
{
    for all ports {
        while (the port is in idle state and there is data to read) {
            read a packet
            if (buffer we just put packet in is over threshold) {
                write buffer to appropriate port
            }
        }
        if (there is any data remaining in the buffer) {
            flush the buffer
        }
    }
}
```

### 4.2.3 Control (passive)

*Passive* control involves responding to an active request in the form of a control packet received by a port. This implementation takes a unified approach: all control response occurs in a localized switch as part of the procedure *doControl()*. A port read procedure, on detecting a control packet, makes a call to doControl() with pointers to the port struct and the packet just received. doControl() examines the control packet and takes appropriate action. An extensive effort was made to avoid distributing control actions throughout the module code, so that debugging and further development would be easier (or, more to the point, possible).

A second phase of control management involves building the table structure presented in section 3.5.2 and implementing the little brother algorithm. As of this writing, system control needs are not sophisticated enough to justify the effort so the little brother algorithm has not yet been implemented.

### 4.2.4 Logging

For debugging and history purposes, each module directs both status and diagnostic messages to a log file. A logfile entry is a single line starting with a timestamp. Following the timestamp is an optional field indicating an error. The rest of the line is a text string that forms the message itself. Log files are created when a module is started, and are named with the following convention: *ddddmmnn* where *dddd* is month/day, *mm* is module type (e.g. ps for PadServer) and *nn* is module number. The utility *ipnstat* is used to retrieve log file information. If a module is started in the foreground from the command line, the argument "-s" will redirect output to the terminal. Here are some example log file entries:

```
[06/02 23:10:03] LINK connected to rainier.1601 from visigoth.1088
[06/03 17:53:40] ERROR: ReadPort: Invalid packet on PS_PORT.12
```

### 4.2.5 Drop-Dead Timers

The system is designed to be reliable, but there are situations in which a module becomes separated from its cluster. This can happen if a module is started when a module using the same ID is already running. For instance, if a user starts Gateway 3 when a Gateway 3 exists, the original Gateway 3 will become a *zombie* when its name database entry is overwritten. Because a module may die unexpectedly for a variety of reasons, the name database does not require an entry to be removed (although modules that terminate gracefully will remove their entries). If a module asks the name database to add an entry, and an entry with the same module ID is found, the existing entry will be overwritten i.e. the name database assumes that the requesting module is properly representing itself and the old entry is no longer wanted.

A "zombied" module will cause no overwhelming problems, but because modules are programmed to live forever, zombies will waste resources and clutter up process tables for an indefinite period of time. More importantly, a zombie PadServer zombies the entire pad cluster; a full text/graphics windowing environment sitting around unused is undesirable. Furthermore, a compute node can host only one text/graphics TypeServer at present. A zombie pad cluster therefore makes that compute node unavailable for another pad cluster.

To combat the zombie problem, each module contains a doomsday device called a *drop-dead timer*. When a module returns from a select() call, it checks its drop-dead timer to see how long it has been sleeping. If the sleep time exceeds a pre-defined period of time and there is still no read activity, the module self-destructs. If not, the drop-dead timer is reset. This mechanism will, of course, cause non-zombie modules to self-destruct, but the timers are set to values larger than the expected maximum idle time of an active module or cluster.

### 4.2.6 Optimizations

As mentioned earlier, the primary design motivator was efficiency. Several small but effective speed optimizations are worthy of mention.

- **Header Prediction:** Packets can be quite small (20 bytes for a pen packet, including the 12 byte header). Therefore, header processing can add significant overhead. Fortunately, because module communication is connection oriented, link-level information included in the core field of every packet is determined at connection set-up and does not change for the duration of the connection. IPN takes advantage of this by pre-assembling packet headers for incoming and outgoing traffic, and including these headers in the port state structure for the connection. For cases where headers need to be generated (when a module buffers packet payload but not the header, or when assembling a control packet), the pre-assembled headers can be copied directly into an outgoing packet rather than setting each header field individually. Macros for setting header fields involve several shift and compare operations, and at least seven macros must be used to completely fill the sync and core fields of a header. Header prediction replaces these macros with two integer assignments. Since the info field of a header is packet specific, this field must be set independently.

A second, and just as important, use of header prediction arises when a packet header has been read and sync has been established. For link-level validation, the port procedure simply compares the core field of the new packet against the core field of the pre-assembled header for incoming packets. In this case, six macro compares are replaced by a single integer compare.

- **Resync Timer**: Although a port read service procedure normally returns after reading a single packet, if synchronization is lost the procedure will read character by character until a packet header is recognized by detection of a sync pattern. This is a rare condition that is usually the result of a programming error or a hardware pad that is transmitting junk. To prevent starvation of other ports, the procedure is allowed to read only a limited number of characters before returning control to the scheduler. The allowable maximum number of characters to read is defined by a *resync timer*. If sync is lost, the timer is set and the procedure begins reading single characters. The timer is decremented by one each time a character is read. When the timer reaches zero, the port goes to sleep if sync has not been restored. This restriction is absolutely necessary to ensure fairness. Resync timers are needed only when byte stream based protocols such as TCP are used. If the low-level communication facility offers message based atomic operations, damaged data will affect only the message in which it is included. This does not mean that damaged data is not a concern; it simply means that synchronization does not depend upon locating a header in an undifferentiated byte stream, because headers are expected to be aligned with message boundaries.

- **Page Alignment:** All IPN buffers are aligned on virtual page boundaries to optimize paging behavior.

## 4.3 Gateway

As a switch, the most important role of a Gateway is to transfer data from one port to another port over many unique port pairs. The Gateway must also efficiently associate port pairs when a connection is requested, and remove associations when connections are dissolved.

The Gateway implementation is built around a central *resource table* that binds together buffers and ports. Paths through the Gateway, called *routes*, are defined by two ports and two buffers. A resource node attached to the resource table contains a pointer to each port structure and a pointer to each buffer. Figure 9 shows the internal structure of a Gateway module.

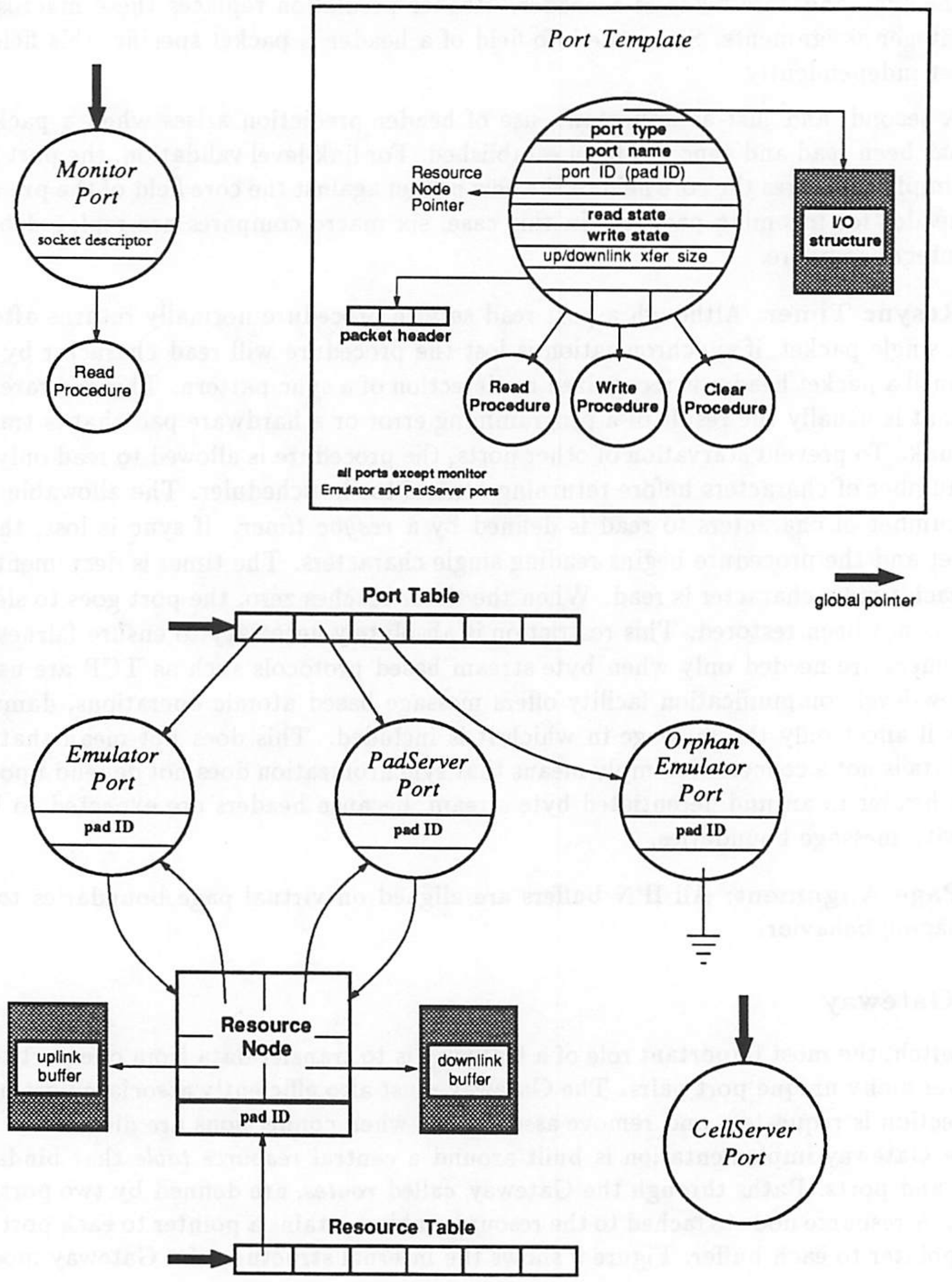

Figure 9: Gateway internal data structures.

The Gateway supports five port types: monitor, CellServer, PadServer, hard link, and soft link (Emulator). A hard link port supports the Ariel hardware interface described in [8]. There is one monitor port and one CellServer port per Gateway. Most routes use a soft link and a PadServer port as internal endpoints.

There are two buffer types: uplink and downlink. Over time, the structure of uplink and downlink buffers has evolved into the same form. The distinction has been maintained in case a need arises that changes the form of one but not the other.

The downlink buffer stores whole packets received from a PadServer port. Headers are not removed in the Gateway because packets in a data stream are interleaved with respect to data type, and the Gateway has no way of reconstructing type from payload. The downlink buffer is written to a link port when the threshold is exceeded as discussed in section 4.2.2. Equivalently, the uplink buffer stores complete packets received from a link port and is written to a PadServer port when the threshold is exceeded. Control packets received on either a link or PadServer port are diverted from the data stream; control packets are most often sent to the CellServer, but the Gateway may choose to dispose of them in some other way.

Global state for a Gateway includes the resource table mentioned above, a port table, a monitor port pointer, and a CellServer port pointer. Most state can be accessed through the resource table, but the scheduler uses the port table because it is a more convenient structure for scanning and because it is the best place to store references to orphan ports, which do not have a route binding.[10]

Procedures for route establishment approximate the pseudocode below. Procedures for route teardown approximate the pseudocode sections for connection failure. As in section 4.2.2, the pseudocode is general. For more detail, see the Gateway source code.

```
a pad contacts the Gateway
if there are sufficient resources {
    create and initialize a link port
    create a resource node, including buffers
    assign pointers between the port and the resource node
    add port to the port table
    add resource node to the resource table
    notify the CellServer that a new pad has arrived
}
else {
    notify the pad that the connection has failed
}
resume normal processing
```

[10]The resource table requires two levels of indirection to access a port structure. The port table requires only one.

If the CellServer has been notified of a pad arrival, it invokes the algorithm described in section 3.5. The CellServer will then report success or failure to the Gateway.

```
if (success and the link port and resource node still exist) {
    create and initialize a PadServer port
    assign pointers between the port and the resource node
    add port to the port table
    notify the CellServer that the connection is successful
    notify the pad that the connection is successful
else {
    detach the link port from the port table
    detach the resource node from the resource table
    free the link port
    free the resource node and buffers
    notify the CellServer that the connection has failed
    notify the pad that the connection has failed
}
```

## 4.4 CellServer

The internal organization of the CellServer is simple in comparison to the Gateway and PadServer, primarily because there are no buffers. However, the present implementation of the CellServer is essentially a shell into which powerful algorithms can be built. The CellServer is a central building block for the InfoPad network system; its critical role as the Gateway controller makes it the proper module for many system control functions. [15] contains a discussion of projected additions to the CellServer. Figure 10 shows the CellServer internal structure.

Global state for the CellServer includes a port table and a pointer to the Gateway port. Much of its functionality has already been described in section 4.2.1. All packets received or transmitted by the CellServer are control, and nothing is stored except port and global state.

The most important job of the current CellServer is implementation of the relocation algorithm described in section 3.5. Starting a PadServer on a remote host is the most complex part of that job. When a pad connection request is forwarded by the Gateway, the CellServer checks the name database for the location of the respective PadServer. If there is no database entry, a new PadServer is created on a compute node defined by a second query to the name database. Equivalently, a new PadServer is created if a message to the address returned by the first query fails because the PadServer has died. CellServers are also required to conduct dialog with other CellServers, under either the same Gateway or a different Gateway.

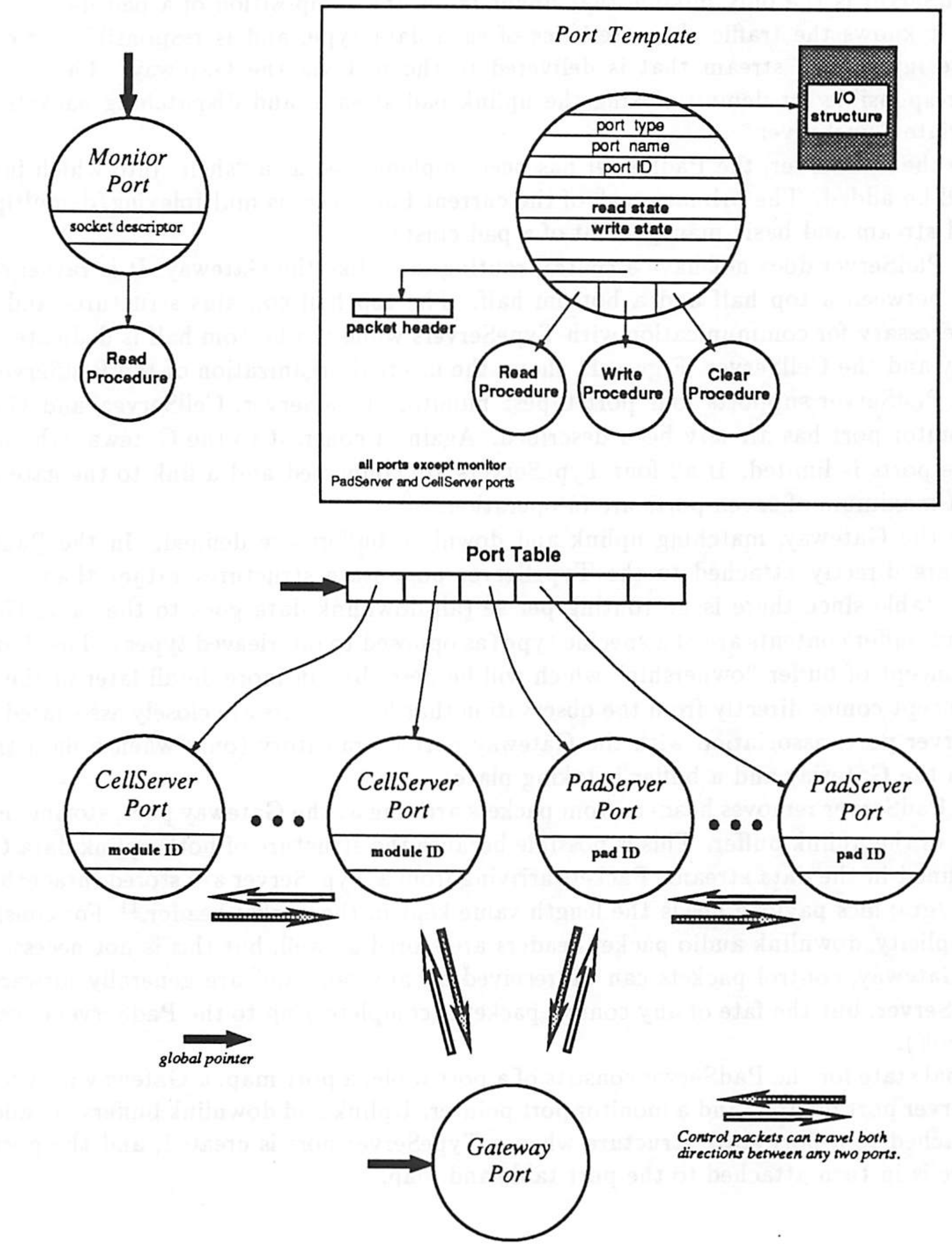

Figure 10: CellServer internal data structures.

## 4.5 PadServer

The PadServer is the only module that understands the composition of a pad data stream in detail. It knows the traffic characteristics of each data type, and is responsible for creating the type interleaved stream that is delivered to the pad via the Gateway. The PadServer is also responsible for demultiplexing the uplink pad stream, and dispatching packets to the appropriate TypeServer.

Like the CellServer, the PadServer has been implemented as a "shell" into which functionality will be added. The primary task of the current PadServer is multiplexing/demultiplexing of a pad stream and basic management of a pad cluster.

The PadServer does not have a central routing table like the Gateway. It is rather sharply divided between a top half and a bottom half. The top half contains structures and procedures necessary for communication with TypeServers while the bottom half is dedicated to the Gateway and the CellServer. Figure 11 shows the internal organization of the PadServer.

The PadServer supports four port types: monitor, TypeServer, CellServer, and Gateway. The monitor port has already been described. Again in contrast to the Gateway, the number of active ports is limited. If all four TypeServers are connected and a link to the gate cluster exists, a maximum of seven ports are in operation.

Like the Gateway, matching uplink and downlink buffers are defined. In the PadServer, buffers are directly attached to the TypeServer port state structures rather than a central resource table since there is no routing per se (all downlink data goes to the same Gateway port) and buffer contents are of a specific type (as opposed to interleaved types). The PadServer has a concept of buffer "ownership" which will be described in more detail later in the paper. This concept comes directly from the observation that buffer pairs are closely associated to one TypeServer port; association with the Gateway port is transitory (only when a data transfer between the Gateway and a buffer is taking place).

The PadServer removes headers from packets arriving at the Gateway port, storing only the payload in the uplink buffer. This is possible because the structure of both uplink data types is fully defined in the data stream. Packets arriving from a TypeServer are stored intact, because the text/graphics payload needs the length value kept in the packet header.[11] For consistency and simplicity, downlink audio packet headers are stored as well, but this is not necessary. As in the Gateway, control packets can be received on any port and are generally forwarded to the CellServer, but the fate of any control packet is completely up to the PadServers version of doControl().

Global state for the PadServer consists of a port table, a port map, a Gateway port pointer, a CellServer port pointer, and a monitor port pointer. Uplink and downlink buffers are allocated and attached to a port state structure when a TypeServer port is created, and the port state structure is in turn attached to the port table and map.

[11]Text/graphics data units contain a 32 bit address followed by a variable length data field. There is no place to encode payload length except in the packet header length field. Uplink data units are fixed length.

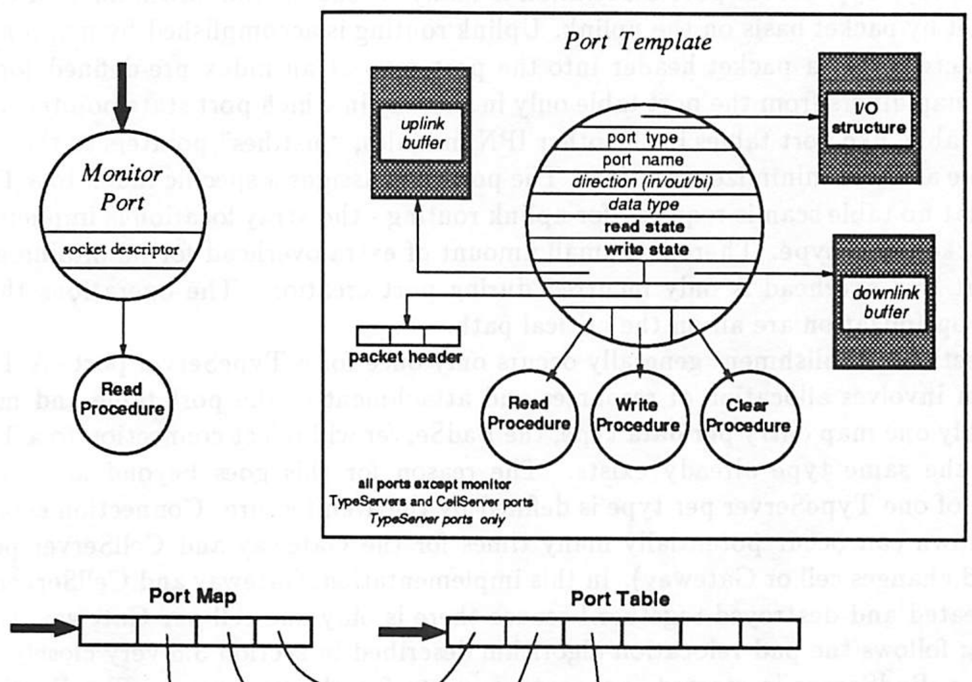

Figure 11: PadServer internal data structures.

There is no direct relationship (i.e. pointers) between a TypeServer port and the Gateway port. Gateway/TypeServer port association is many to one on the downlink, and determined on a packet by packet basis on the uplink. Uplink routing is accomplished by mapping the data type extracted from a packet header into the port map at an index pre-defined for the type. The port map differs from the port table only in the way in which port state pointers are stored. The port table, like port tables in the other IPN modules, "justifies" pointers to the low indices in the table array to minimize scan time. The port map assigns a specific index to a TypeServer port so that no table scan is required for uplink routing - the array location is implicitly defined by the packet data type. There is a small amount of extra overhead for maintaining two port tables, but this overhead is only incurred during port creation. The operations that benefit from this optimization are all on the critical path.

Connection establishment generally occurs only once for a TypeServer port. A TypeServer connection involves allocation of resources and attachment to the port table and map. Since there is only one map entry per data type, the PadServer will reject connection to a TypeServer if one of the same type already exists. The reason for this goes beyond array space; the limitation of one TypeServer per type is defined by the architecture. Connection establishment and teardown can occur potentially many times for the Gateway and CellServer ports (each time a pad changes cell or Gateway). In this implementation, Gateway and CellServer ports are always created and destroyed together because there is only one cell per Gateway. Connection remapping follows the pad relocation algorithm described in section 3.5 very closely.

When a PadServer is started, no context exists for the pad user. The PadServer first establishes monitor and Gateway ports, and then waits for a CellServer connection request on the monitor port. After a CellServer connects (the one which started the PadServer if the PadServer was autostarted), the PadServer attempts to autostart essential user state: a text/graphics TypeServer and a pen TypeServer. If successful, the PadServer connects to the Gateway specified by the CellServer cluster number. Finally, the PadServer sends acknowledgement to the CellServer and begins to forward text/graphics data to the Gateway. The user now has a window environment and an input device. TypeServer autostart follows the same procedure as PadServer autostart.

Several other aspects of the PadServer deserve mention because they are either unique or essential:

- **Link Locking:** Early implementations of the PadServer exposed a serious flaw in the basic design through two related pathological conditions: **1)** PadServers that had been idle long after their drop-dead timers should have expired continued to run, and **2)** a Gateway (link) port blocked for any reason caused the PadServer to consume about 95% of CPU time on otherwise idle compute nodes.[12]

  In the first case, investigation revealed that the Gateway port had been disconnected but, even though there was no activity from the pad, the text/graphics TypeServer was still

[12]In this case, "blocked" refers to any inability to write on the Gateway port, not the act of sleeping on a system call as discussed earlier. A Gateway port can become blocked if the communication protocol in use is reliable and the pad refuses to consume data sent from the Gateway. That is, IPN modules will write as much as possible and then stop if the recipient refuses to read, rather than discard the data.

periodically sending small screen updates. The PadServer would service the input and reset the drop-dead timer, so the timer would never expire.

In the second case, a busy-wait condition had been entered after the Gateway port became blocked. When a downlink buffer filled and the PadServer could not write to the Gateway, the TypeServer input request could not be satisfied. The select() system call would continue to invoke the scheduler in an attempt to satisfy the request.

The fix for this problem is the *link lock*. If the Gateway connection is terminated, all TypeServer ports are removed from the scheduler port scan making the Gateway port unavailable to a TypeServer. The Gateway port is locked whenever the PadServer is unable to write to it e.g. when the connection disappears or when a write fails on an existing connection. Locks do not affect the monitor or CellServer ports. Also, the lock applies **only** to the Gateway write side.

The term "link" in the PadServer context refers to the PadServer-Gateway connection. It has no relationship to Gateway hard and soft links. Link locking is most important in the PadServer, but has been implemented in the Gateway because it is useful there as well and presents only a small load.

- **Text/Graphics Data Loss**: This topic is related to link locking, and is important to avoid startup failure. On text/graphics TypeServer startup, link locking is disabled until the initial screen refresh completes. If the Gateway connection is down, any text/graphics data received during this period is discarded. This measure is necessary because the text/graphics TypeServer is unavailable to its clients until the screen dump is complete, and the pen TypeServer will quit if it fails to contact the text/graphics TypeServer within a short period of time. Therefore, the entire pad cluster startup will be incomplete unless the text/graphics TypeServer is allowed to write. No data is lost, because the text/graphics TypeServer maintains current screen state. The screen can be updated at a later time.

- **Partial Writes on the Gateway Port**: The Gateway port write procedure can be passed any of the four TypeServer downlink buffers on any invocation. If the Gateway connection is implemented with a byte stream protocol such as TCP, there is no guarantee of write atomicity for a packet within a buffer. If the buffer is partially written (the Gateway port enters a sleep state), a packet may be truncated. If the next write on the Gateway port is not from the same buffer, several packets may be destroyed[13] and the Gateway will spend valuable time restoring synchronization. The problem is solved in two ways: First, if a partial write occurs but there is no kernel error (some but not all of the kernel buffer was sent to the network), the unwritten buffer contents are shifted to the buffer head and the write immediately retried. Second, if the kernel returns a system error on a write retry, the link is locked. The **only** way to unlock the link is to complete the partial buffer write successfully.

[13]The truncated packet, the first packet in the new buffer, and the next packet in the first buffer. They are chewed up by resynchronization.

- **Buffer Ownership**: As discussed previously, buffers are closely associated with TypeServer ports. The ports are said to "own" their buffers. However, buffer ownership is transferred to the Gateway port in the write procedure, and the Gateway port stores a pointer to the buffer in its state struct. If a partial write occurs and the respective TypeServer port closes before the buffer can be completely written, the close procedure will detect that ownership has been reassigned. The close procedure will close the port, but will not free the downlink buffer. When the Gateway port has successfully written the remainder of the buffer, it checks to see if the port has been closed. If so, it discards the buffer. Although TypeServer port closure is expected to be rare, management of ownership completes a set of mechanisms that ensure correct packet delivery to the Gateway. If the packet sync pattern is ever removed (reducing the header overhead by one third) this mechanism will be absolutely essential if the PadServer is to operate properly with all communication protocols.

## 4.6 Library

The library is the glue that holds InfoPad modules together. It is included into every module in the prototype. The library is organized into three sections: the custom I/O package, the name database, and the connect/control library. These sections have a hierarchical relationship with each other, but the primary entry points of each section are available to all modules. So, from the point of view of a module, each section is a library in its own right. Figure 12 shows the relationship between sections of the library. The arrows in the figure show direction of call invocation, not data flow.

The library is not a module, and does not contain the common elements outlined in 4.2. It is a distributed body of code statically compiled into modules that make use of its services.

### 4.6.1 Custom I/O Package

Because system calls are expensive, buffered I/O is essential to efficiency. Buffered I/O allows a user level process to read entire system buffers into user space in a single call,[14] and then meter data to the program through relatively efficient (in comparison to a system call) byte copies. A single buffered I/O read often gathers multiple packets. In IPN, this is extremely important since a packet read operation actually consists of two reads: one for the header (12 bytes) and one for the payload. Module buffers are not well suited to buffered I/O because something must be known about the packet before data is written to a module buffer. This "something" is usually contained within the packet header. The I/O buffer will contain everything passed upward by the system, while module buffers should contain only "good" data and, in some but not all cases, headers.

[14]This is not completely accurate, since the amount available may be larger then the buffered I/O storage space.

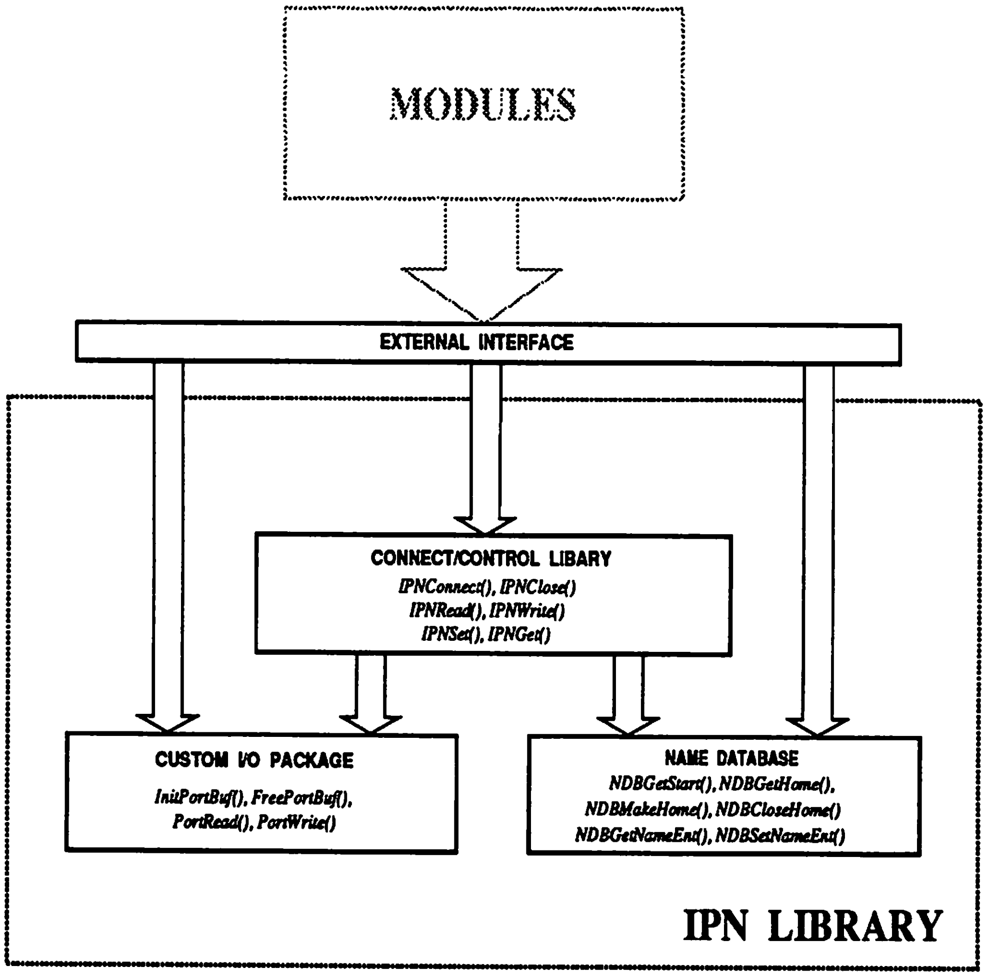

Figure 12: Delivery support library subsections.

Most operating system releases include buffered I/O packages. However, IPN has several special requirements not met by standard I/O packages: **1)** Module programming is considerably simplified if the buffered I/O read procedure is looks atomic i.e. when a read is issued that requests N bytes, either N bytes are returned or zero bytes are returned. For instance, if a module tries to read from a port and the read is not atomic, the module code must be prepared to handle partial reads. It must be capable of storing the data it acquired and restarting the read at some offset on the next try. This can get quite ugly. **2)** The I/O package must be flexible. For instance, IPN I/O buffer structures include small counters for execution tracing. These counters can be modified without affecting the normal operation of a module. Other structure fields may be added or removed to meet the changing needs of the system. **3)** Generic I/O packages often contain functionality not needed by IPN. In accordance with the principle of frugality [16], a custom I/O package will contain only the function required, thus allowing a module to be smaller and meaner.

Entry point procedures for the buffered I/O package are described below. As mentioned previously, these procedures are available to any module that includes the library. However, direct use is discouraged. The connect/control library defines macros for each procedure; since the connect/control library contains communication protocol specific code, use of macros helps to ensure library compatibility.

**I/O procedures:**

**InitPortBuf** (fd)
Initialize buffer structure. *Fd* is the Unix socket descriptor for a port. The procedure returns a pointer to an I/O structure. This pointer must be included as an argument with all future read and write actions on the port.

**FreePortBuf** (fi)
Free the I/O structure. *Fi* is the pointer returned by InitPortBuf().

**PortRead** (buf, sitems, qitems, fi)
Read *qitems* of size *sitems* into *buf* from the port specified by *fi*. Returns qitems or zero. This procedure is modeled after the Unix buffered I/O function *fread*().

**PortWrite** (fi, buf, qreq)
Write *qreq* bytes from *buf* to the port specified by *fi*. Returns the number of bytes written or error. Unlike PortRead, this procedure is not atomic.

### 4.6.2 Name Database

The name database offers a way to transparently map ID triples into network addresses. It is currently implemented in a limited form: data is stored in files, and a module running on a compute node that does not mount the file system containing the database files cannot use the database. A dynamic network-based name server is in the planning stages. The name server will have essentially the same functionality as the name database, at least at first.

The name database maintains two sets of data: the ***startup set*** and the ***run set***. The startup data set contains the desired start-up location of several text/graphics TypeServers, pen TypeServers, and PadServers. Gateways and CellServers are not included in the startup set because they are manually started by invoking one of the IPN utilities. Listing all possible entries would be difficult (3 x 256 = 768), so a default entry is included for all module numbers not included. Here is a sample entry from the startup set:

**ts 4 15 yellowstone.eecs.berkeley.edu 0**

The first three fields contain the modules ID triple. This entry indicates that the pen TypeServer for pad 4 should be started on yellowstone. The number 15 is the module number for pen TypeServers, which is also its data type (0x0f). The fourth field is an unused placeholder.

The run data set contains the network locations of currently running modules. Here is a sample entry from the run set:

**gw 0 49 visigoth.eecs.berkeley.edu 1232**

This entry shows that Gateway 49 is running on visigoth at network port 1232 (network ports are not equivalent to module ports). The module number (third field) is zeor if the entry refers to a Gateway or PadServer. The run set contains entries only for modules that accept connections, namely the Gateway, the PadServer, and the CellServer. The location of TypeServers can be determined from the startup data set. Limiting membership in the run set reduces contention for database files (modules must wait on a lock for about 50ms if the file is in use). The format of entries in both data sets is identical to allow manipulation by the same procedures.

As with other sections of the library, name database access points are directly accessible by modules. Gateways and PadServers call one name database procedure on startup to register themselves and another procedure when they exit to remove their run set entry. Otherwise, name database procedures are typically invoked by the connect/control library as part of connection establishment. For instance, if a CellServer wishes to connect to a PadServer, it calls a single function in the connect/control library with the module ID of the PadServer (among other things) as an argument. The connect/control section invokes the name database on behalf of the CellServer, and manages all tasks required to complete a connection.

**Name database entry points:**

**NDBGetStart** (moduletype,modulenum,clusternum, hostname)
Get the starting location of (*moduletype,modulenum,clusternum*) and return in *hostname.*

**NDBGetHome** (moduletype,modulenum,clusternum, hostname,hostport)
Get the current location of (*moduletype,modulenum,clusternum*) and return in (*hostname,hostport).*

**NDBMakeHome** (moduletype,modulenum,clusternum, hostname,hostport)
Register (*moduletype,modulenum,clusternum*) at (*hostname,hostport)* in the run set.

**NDBCloseHome** (moduletype,modulenum,clusternum)
Remove the run set entry for (*moduletype,modulenum,clusternum*).

**NDBGetNameEnt** (datafile, lockfile, ep)
Get an entry from *datafile* using advisory lock file *lockfile* that matches the entry structure *ep.* Return in *ep.*

**NDBSetNameEnt** (datafile, lockfile, ep)
Create an entry in *datafile* using advisory lock file *lockfile* using data in structure *ep.*

NDBGetNameEnt() and NDBSetNameEnt() are generalized procedures that operate on both data sets. Other NDB procedures use the services of NDBGetNameEnt() and NDBSetNameEnt().

### 4.6.3 Connect/Control (active)

The connect/control section of the library implements the interface defined in section 3.4.3. It forms the top layer of the library hierarchy. The term *active* means that the Control/Connect library is the *initiator* of control actions. The module procedure doControl() described in section 4.2.3 implements the passive side of module control.

The term connect/control indicates the functional division of the section. Connect procedures are responsible for establishing and terminating connections, and providing packet-oriented I/O operations using the custom I/O package. Control procedures are responsible for hiding operations on packet data structures. Connect and control procedures together provide an abstraction that simplifies implementation of low-level communication protocols. A detailed description of a study in which three protocols were installed into the prototype and tested is presented in [9].

As described in section 3.4.3, the library defines an external interface and an internal interface. The distinction lies in policy more than mechanism, and is realized through procedure groupings. Any of the procedures available to one interface are available to the other, but the formal (programming) definition of procedures intended for use with the external interface are not subject to unannounced changes. Modules using the external interface (e.g. TypeServers) can be confident that the procedures will always behave as specified. The internal interface, on the other hand, may be changed at any time. Procedures included in both interfaces are governed by the rules for external interfaces.

---

**Procedures included in both interfaces:**

**IPNConnect** (smt,smn,scn, dmt,dmn,dcn, isnonblock, rdpkts, wrpkts)
*(smt,smn,scn)* defines the source module ID, *(dmt,dmn,dcn)* defines the destination module ID, *isnonblock* is a flag that is true if the connection should be non-blocking, and *rdpkts/wrpkts* are flags that are true if the caller wants to read or write IPN packets. The default is a depacketized byte stream. Generally, a TypeServer (or Emulator) will use a macro to simplify the call. For instance, a pen TypeServer might use the command PenConnectToPS(padid). IPNConnect() returns a pointer to an I/O structure.

**IPNRead** (ptr, size, nitems, fi)
*Ptr* is a pointer to the destination buffer, *size* is the size in bytes of the object to read, *nitems* is the number of objects to read, and *fi* is an I/O structure pointer. The semantics of IPNRead are identical to PortRead() - in fact, IPNRead() is built on top of PortRead().

**IPNWrite** (ptr, size, nitems, fi)
These parameters have the same meaning as those for IPNRead, except that ptr points to a buffer containing data to be written on the channel. The semantics of IPNWrite() are modeled after the Unix buffered I/O function *fwrite*(). IPNWrite() is built on top of PortWrite().

**IPNClose** (fi)
This call closes the connection referenced by *fi* and frees all associated resources.

Most of the following procedures follow a "4-phase" pattern. In the first phase, a control packet is dispatched to a remote module. In the second, the remote module receives and processes the packet. The remote module acknowledges the sender in the third phase, and the sender processes the acknowledgement in the fourth. The first and fourth phase correspond to the *active* participant (the module that initiated the dialog) while the second and third phases correspond to the *passive* participant.

The first and third phases implement a 2-way request-reply handshake procedure. This procedure is essentially the common 3-way handshake where the third phase is implied under TCP. If UDP or another unreliable transport protocol were used, an explicit third "way" would be needed.

Currently, eight operations are defined for generalized message passing. These operations are implemented by the following library calls.

**IPNSendGetReq** (fi, msgtype, code, srcID, dstID)
**IPNProcGetReq** (fi, hdr, msgtype, code, srcID, dstID)
**IPNSendGetACK** (fi, msgtype, msglen, msgdata, srcID, dstID)
**IPNProcGetACK** (fi, hdr, msgtype, msglen, msgdata, srcID, dstID)

*Fi* is a pointer to the port I/O structure, *hdr* is a pointer to the header of a packet received on the port indicated by fi, *msgtype* is a user-defined constant that is interpreted in an arbitrary (but consistent) way at the endpoints, *msglen* is the length of a payload, *msgdata* is a payload, *code* is an arbitrary result code, and *srcID* and *dstID* are the endpoint source and destination ID triples encoded into a single value. A module requiring data from another module sends IPNSendSetReq(), and the receiving module replies with IPNSendSetACK(). The requested data is stored in the payload. IPNProcGetReq() and IPNProcGetACK() extract information from their respective packet types and return the information via value-result parameters. Control data will typically be a structure cast to a character string.

**IPNSendSetReq** (fi, msgtype, msglen, msgdata, srcID, dstID)
**IPNProcSetReq** (fi, hdr, msgtype, msglen, msgdata, srcID, dstID)
**IPNSendSetACK** (fi, msgtype, code, srcID, dstID)
**IPNProcSetACK** (fi, hdr, msgtype, code, srcID, dstID)

Interpretation of the arguments for Set procedures is identical to that for Get procedures. IPNSendSetReq() requests that the destination module store *msgdata* in some pre-defined location. The destination responds with IPNSendSetACK(). IPNSendSetReq() and IPNProcSetReq() are duals of IPNSendGetACK() and IPNProcGetACK(), respectively, while IPNSendSetACK() and IPNProcSetACK() are duals of IPNSendGetReq() and IPNProcGetReq(). The effect of Set and Get actions is similar: in the first case, the active module produces the payload. In the second case, the payload originates from the passive module.

Procedures defined for the internal interface only are generally intended to support connection procedures and inter-module utilities.

**Procedures included in the internal interface only:**

**IPNQckConnect** (smt,smn,scn, dmt,dmn,dcn, isnonblock,rdpkts,wrpkts)
This procedure offers the same function as IPNConnect, but does not perform a connection handshake. This leaves the destination process (Gateway or PadServer) in an intermediate "registered but not formally connected" state for the calling process, allowing relaxed connection semantics for administrative tasks.

**IPNSendConnReq** (fi, smt,smn,scn, dmt,dmn,dcn)
**IPNProcConnReq** (fi, hdr, callermt,callermn,callercn)
**IPNSendConnACK** (fi, code)
**IPNProcConnACK** (fi, hdr)
As before, *(smt,smn,scn)* and *(dmt,dmn,dcn,)* refer to the source and destination modules, and *fi* is a pointer to the I/O struct. *Code* is a context specific value. IPNSendConnReq() sends a connection request to the destination module. IPNProcConnReq() processes a connection request by verifying source and destination, and forms the pre-assembled headers for the destination port. *Hdr* is the received connection request packet and *(callermt,callermn,callercn)* is the module ID of the caller. IPNSendConnACK() sends a connection acknowledgment to the active module. *Code* represents the result of the connection attempt: 0 for success and -1 for failure. IPNProcConnReq() processes a connection request acknowledgment by verifying source and destination, and forms the pre-assembled header for the source port.

**IPNSendTermReq** (fi, code)
**IPNProcTermReq** (fi, hdr)
**PNSendTermACK** (fi, code)
**IPNProcTermACK** (fi, hdr)
Send and process a terminate signal. Terminate signals are designed to terminate pad and gate clusters. Terminate requests are sent to Gateways and forwarded to CellServers. In IPNSendTermReq(), *code* is a value that specifies the extent of the action: 0 means kill a pad cluster and 1 means kill the gate cluster. In IPNSendTermACK(), *code* is the result of the terminate action. IPNProcTermReq() and IPNProcTermACK() both return *code* from the packet represented by *hdr*.

**IPNSendFwdReq** (fi, code)
**IPNProcFwdReq** (fi, hdr)
**IPNSendFwdACK** (fi, code)
**IPNProcFwdACK** (fi, hdr)

These procedures are used only by the Gateway and the CellServer. Their semantics are identical to the terminate request procedures, but instead of propagating a terminate request they propagate a connection request from a pad to the CellServer.

**IPNSendPrbReq** (fi, code)
**IPNProcPrbReq** (fi, hdr)
**IPNSendPrbACK** (fi, code)
**IPNProcPrbACK** (fi, hdr)

Send a probe to the module referenced by *fi* with a parameter in *code*. The receiving module will return an acknowledgment immediately if it is still alive. The semantics of these procedures are identical to the semantics for the terminate set. Any module can send a probe packet.

**IPNSrcQnch** (fi, code)

Ask a connected module to stop sending data. This is an advisory message: the receiving module should stop if it can, but is not required to do so. It indicates that resources in the sending module are limited and that data may be dropped. The acknowledgement for a source quench is the cessation of data flow.

**IPNSrcResm** (fi, code)

Inform a "quenched" module that data flow can resume. Acknowledgment of this message is a data packet arriving at the port.

## 5 Implementation of Type Support, an Application, and the Emulator

This section presents a brief summary of the type support service group, one application, and the pad Emulator. Figure 13 is a version of figure 2 showing the modules discussed in this section with interconnections. All processes are relocatable, but text/graphics and pen TypeServers are run on the same network host by convention so that location of active InfoPad processes is made easier, and because both the text/graphics and pen TypeServers have host specific hard-wired parameters.

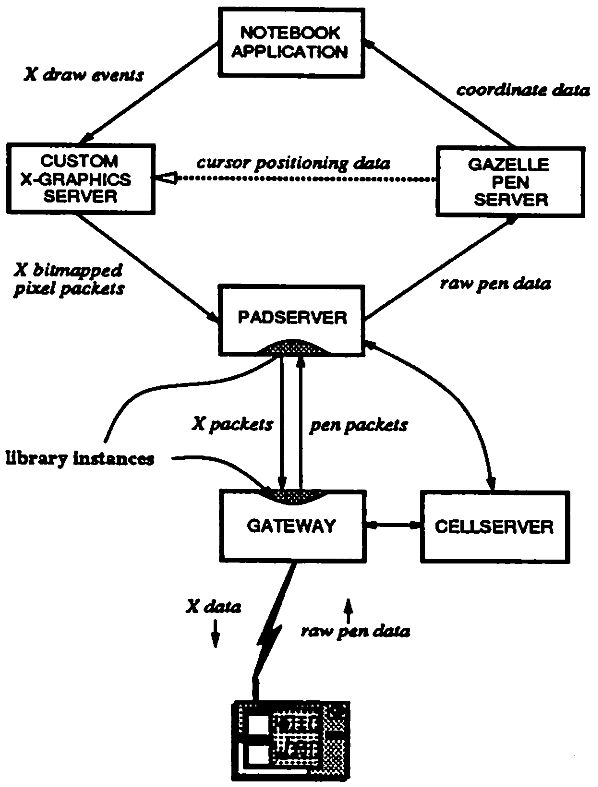

Figure 13: InfoPad network topology: modules and data flow for pen loop-back.

### 5.1 TypeServers

At present, two type support modules have been developed to support a basic user environment: a text/graphics TypeServer and a pen TypeServer. The pen TypeServer receives pen packets from a PadServer, performs some data conversion on the pen data, and transfers the data to the text/graphics TypeServer as cursor positioning information or to the application as inking data. The text/graphics TypeServer provides a window based user interface.

### 5.1.1 Text/Graphics

The text/graphics TypeServer is implemented with a modified version of an X-Windows R5 server. The server is "split" above the frame buffer: data normally sent to the frame buffer is packetized by an instance of the IPN library and sent to the PadServer. The packet payload is an address/data pair where the address is the starting X-Y location on a 480x640 monochrome screen and the data is a set of 32 bit integers, each bit representing a screen pixel.[15] The server is modified only from the viewpoint of the PadServer; all other modules and applications communicate with the server through an interface defined by the X Consortium, normally by sending standard X events. The X server is an excellent example of the abstraction hoped for in TypeServers. An X client connected to the server does not have to know that its terminal device is an InfoPad. The X server appears exactly as if it were controlling the monitor on a local host. The X server places one restriction on InfoPad network topology: there can be no more than one InfoPad X server per host, because the server number of the X based TypeServer is hard coded. This will be corrected in the future.

### 5.1.2 Pen

The pen TypeServer is a custom X client with an IPN external interface and a specialized interface for the application. The pen server manages pen data for several different types of hardware pads, including the Gazelle pad used for all performance measurements. For Gazelle, data arriving at the pen TypeServer from the PadServer is a depacketized stream of five byte data units. The first byte indicates a button press (the pen used with the Gazelle pad has one button at the tip and one on the side) and the last four record an X-Y coordinate. The pen TypeServer extracts and remaps the X-Y data into a different coordinate system and applies a mask to the button byte before forwarding the data in a similar form to the X server. If an application is connected, a copy of the data is also sent to the application.

A simple video TypeServer has been constructed to deliver VQ encoded video data to a prototype chip set via the hardware interface supported by the Gateway. This server is a simple program that transfers the contents of a file to an IPN interface at a more-or-less metered rate. It is useful only for testing purposes, but has been included here for completion. Currently, there is no audio TypeServer.

[15]The X server calls IPNConnect with the wrpkts bit reset so that the library performs packetization. Since data boundaries are maintained on IPN writes, an address/data pair can be sent in a single IPNWrite call with no delimiter. It will be placed in its own IPN packet.

## 5.2 Application

The only application written specifically for InfoPad is an X client called *Notebook*. This application generates a graphics window with a region that resembles a paper note pad. If a user draws in the window with a pen or a mouse, electronic ink appears under the pen on the screen. Notebook is an instrumental part of the measurement system, since pen loop-back latency is a critical system metric [8]. Notebook sends inking information to the X server using X events. It receives pen information from the pen TypeServer via a TCP connection.

## 5.3 Emulator

The Emulator is implemented as an X client, like the pen TypeServer. However, in contrast to the TypeServer, the Emulator contacts the X server that controls the console screen on the host where Emu was started. So, the Emulator has at least two connections: one to the local X server and one to a Gateway. The Emulator displays the X environment created by the InfoPad X server in a window on a local workstation. The window is intended to "emulate" the screen of the hardware pad. Emulators can be nested i.e. an Emulator can be started from an xterm inside an Emulator window. Fairly complex module interconnection can be generated in this way.

It is expected that the Emulator will be modified extensively to support TypeServer development. Most TypeServers will have an end-to-end view with the TypeServer itself on one end and a pad on the other. The Emulator can be programmed to emulate pad behavior under may different conditions.

## 6 Conclusions

This system represents the first serious attempt to construct a working InfoPad network prototype. It is a beginning from which, it is hoped, a widely used technology with a rich set of capabilities will grow. If the work presented here is successful, system growth and maturation will be possible the template architecture.

The prototype is currently in everyday operation, and application developers are beginning to use the programming model in varied and aggressive ways. As expected and welcomed, these developers are exercising weak points in the implementation, so the prototype is undergoing frequent minor change. These changes are typically related to unanticipated needs and removal of inflexibilities. For example, although the InfoPad display will be black and white for quite some time, one developer requires the ability to autostart either color or black and white X servers in support of advanced work. This need will result in a further generalization of parameter passing in IPN. So far (and surprisingly), only one problem has been detected that can be blamed on a coding bug.

Although the body of practical experience is still small, enough insight has been gained to at least begin reviewing the system goals.

- **Speed**: At its best, the system appears to be fast. An X screen update completes in about a half second, and cursor movement is nearly as rapid as on the console. There is an unacceptable degree of variation in performance, however. At worst, screen updates may take several seconds (in bursts), and cursor movement is delayed. Average performance is difficult to identify; the difference between "good" and "bad" performance is almost exclusively related to pad ID, which implies that network host assignment is more important than expected. To clarify, each pad ID represents a specific set of machines for the PadServer, text/graphics TypeServer, and pen TypeServer. One pad ID may generally run fast while another may generally run slow.

- **Efficiency**: About the only comment that can be made at present regarding efficiency is that operation of processes not related to the prototype do not appear to be affected by the presence of InfoPad processes. The developers have not noticed any affect, and there have been no complaints from others.

- **Simplicity**: If anything, the implementation became simpler as work progressed. Many common elements emerged that could be applied across several modules, and insistence on simple structures and algorithms was not diluted. As a result, code is not unduly complex and should be easy to understand by those new to the system.

- **Reliability**: As mentioned above, only one bug has been identified, and that bug is exercised only under specific conditions. Modules have not been mysteriously dying or mangling data. Log files have revealed some unsuspected behavior, but the system is robust enough to self-correct.

- **Concurrency**: So far, very little is known about the effect of concurrency measures. The complete lack of pathological conditions when multiple pads are in operation seems to indicate that the time-sliced/non-blocking approach is operating as intended. That is, the system does not "hang" waiting for an event, and service intervals appear to be load dependent and uniform across pads rather than related to specific pad activity.

- **Fairness**: Like concurrency, this goal is difficult to quantify at present since no attempt has been made to take advantage of scheduling capabilities. There is no reason to expect that fairness is violated, because it is enforced by the first-come-first-served nature of the scheduler.
- **Flexibility**: The system has already proven to be flexible. Modifications have been easy to make, and the learning curve for new module developers has been short mainly because the structure allows modularization of tasks.

Some of these goals will be aided by system evolution. For example, the present move to Solaris on ATM will potentially remove network induced bottlenecks.

Work is in progress on the next version of the architecture. The central element that distinguishes the next generation from that presented here is the use of *proxy* connections: instead of directing all data for a pad through the PadServer, TypeServers will connect directly to the Gateway. The PadServer will continue to act as "proxy" control for the individual connections. This will require a far more sophisticated method of connection management than presently exists. [15] gives an overview of the next generation architecture.

Two important aspects of the current architecture and implementation are explored in [8] and [9].

The first describes a detailed study of pen loop-back latency, including measurements under varied system load. This study found that loop-back latency (the time between pen contact and the appearance of ink on the display) is approximately 10ms with pen data only (pen packets on the uplink and X packets on the downlink). With addition of a video load, the tail of the approximately exponential distribution extends somewhat, but the mode remains fairly stationary. The paper also compares the performance impact of various low-level protocols.

The second reference discusses an investigation into low-level communication protocols for InfoPad. It presents a survey of existing methods, describes the implementation of three protocols, and reviews a set of measurements. In part, this paper concludes that packet filters are an attractive alternative to heavyweight transport/network layer protocols.

In summary, this paper has described an extensive effort into building infrastructure for the InfoPad project. The prototype will continue to evolve and grow throughout the lifetime of the project. The ultimate goal of this work is to facilitate the learning process, and to demonstrate that InfoPad is a promising and realizable concept.

## 7 Acknowledgements

There are many people responsible for the continued success of the prototype. I would particularly like to thank my academic advisor Professor Robert Brodersen and the Infonet group leader Professor Jan Rabaey. The architecture was developed primarily by the Infonet group whose members include Richard Edell, My Le, Chris Long, Ken Lutz, Shankar Narayanaswamy, Brian Richards, and Srinivasan Seshan. These people form the brain trust for wired network development. I would also like to thank our system administrator Kevin Zimmerman for his cheerful and competent support despite our insistent and never-ending demands.

## References

[1] M. Accetta, et al., "Mach: A New Kernel Foundation for UNIX Development," in *Proceedings of the Summer Usenix*, July 1986.

[2] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism," in *Proceedings of the 13th SOSP, Operating Systems Review*, Vol. 25, No. 5.

[3] B.N. Bershad, et al., "Lightweight Remote Procedure Calls," in *ACM Transactions on Computer Systems*, Vol. 8, No. 1, Feb 1990.

[4] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy, "User Level Interprocess Communication for Shared Memory Multiprocessors," in *ACM Transactions on Computer Systems*, Vol. 8, No. 2, May 1991.

[5] A.D. Birrell, R. Levin, R.M. Needham, and M.D. Schroeder, "Grapevine: An Exercise in Distributed Computing," in *Communications of the ACM*, Vol. 25, No. 4, April 1982.

[6] A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," in *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984.

[7] P. Brinch-Hansen, "The Nucleus of a Multiprogramming System," in *Communications of the ACM*, Vol. 13, No. 4, April 1970.

[8] F.L. Burghardt, "Measurement and Optimization of Inking Latency in a Pen Pad Based Computer System," *EECS266 project report, University of California at Berkeley*, May 1994.

[9] F.L. Burghardt, "Protocols for a Multimedia System: Investigation into Transport Mechanisms for the InfoPad Network," *EECS262 project report, University of California at Berkeley*, May 1994.

[10] L.F. Cabrera and D.E. Long, "Swift: Using Distributed Disk Striping to Provide High I/O Data Rates," in *Computing Systems*, Vol. 4, No. 4, Fall 1991.

[11] D.R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel," in *ACM Transactions on Computer Systems*, Vol. 3, No. 2, May 1985.

[12] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," in *IEEE Communications Magazine*, June 1989.

[13] D.E. Comer and D.L. Stevens, *Internetworking with TCP/IP*, volumes I-III. Englewood Cliffs, N.J.: Prentice-Hall, 1991.

[14] H.C. Lauer and R.M. Needham, "On the Duality of Operating System Structures," in *Proceedings of the 2nd SOSP and Operating Systems Review*, Vol. 13, No. 2, April 1979.

[15] M.T. Le, S. Seshan, F.L. Burghardt, J. Rabaey, "Software Architecture of the InfoPad System," submitted to *Workshop on Mobile and Wireless Information Systems (MOBIDATA)*, Rutgers University, October 24/25, 1994.

[16] H. Massalin and C. Pu, "Threads and Input/Output in the Synthesis Kernel," in *Proceedings of the 12th SOSP, Operating Systems Review*, Vol. 23, No. 5, December 1989.

[17] J.K. Ousterhout, D.A. Scelza, and P.S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," in *Communications of the ACM*, Vol. 23, No. 2, Feb 1980.

[18] D.D. Redell, et al., "Pilot: An Operating System for a Personal Computer," in *Communications of the ACM*, Vol. 23, No. 2, Feb 1980.

[19] M.D. Schroeder, A.D. Birrell, and R.M. Needham, "Experience with Grapevine: the Growth of a Distributed System," in *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984.

[20] W.R. Stevens, *Unix Network Programming*, Prentica Hall Software Series, Englewood Cliffs, N.J.: Prentice-Hall, 1990.

[21] A. S. Tanenbaum, *Computer Networks.* Engelwood Cliffs, N.J.: Prentiss-Hall, 1989.

[22] J. Walrand, *Communication Networks, a First Course.* Boston, Ma.: Irwin, 1991.

[23] W.A. Wulf, et al., "Hydra: The Kernel of a Multiprocessor Operating System," in *Communications of the ACM*, Vol. 17, No. 6, June 1974.