An Architecture for Call Processing

See-Mong Tan and Roy H. Campbell
Department of Computer Science,
University of Illinois at Urbana-Champaign,
Urbana, IL 61801.

Anand Iyengar and Ravi Sethi
AT&T Bell Laboratories,
600 Mountain Avenue,
Murray Hill, NJ 07974.

May 22, 1991

Abstract

Call processing is concerned with connection setup, connection teardown, channel allocation and channel management for data switches in a circuit or virtual circuit-based network. *Archos* is a software architecture for call processing. The goals are flexibility, portability, switch independence, scalability and extensibility in the software. *Archos* is written as a C++ application of the *Choices* object-oriented operating system framework. The object-oriented approach provides consistent partitioning of call features, easy maintenance and simple prototyping of new features. Because *Choices* is a multiprocessor operating system, parallel processing can scale performance of the switch controller as utilization grows.

1 Introduction

Call processing is concerned with connection setup, connection teardown, channel allocation and channel management for data switches in a circuit or virtual circuit-based network. Call processing software controls the circuit or data switches in the network, thus such software systems also function as switch controllers. The terminology of the data switching field borrows heavily from telephony, where call-processing originated. Establishing a connection in virtual circuit networks is very similar to placing a POTS ("plain old telephone service") call. A user on a workstation wanting to connect to another remote computer across such a network executes a connection establishment procedure. Setup negotiations take place between the source machine and its local switch, from the local switch to other switches within the network until the destination machine is reached. At each switch, call processing software handles the necessary setup negotiations for the call. While the actual switching of packets is done at the hardware level, connection establishment, route selection[1], channel allocation and connection teardown is the responsibility of the call processing software.

[1] In source routed networks, route selection is the responsibility of either the initiating host or the first switch in the virtual circuit path.

The lack of a coherent guiding architecture for the software can lead to a "corrupted" software architecture, including inconsistent partitioning of call features, high maintenance costs, high development times and "breakage" of existing features with the introduction of new ones — problems which have plagued AT&T's No. 5ESS[DH82] telephone switch software [A+86]. Instead, organizing the controller software around the dynamic nature of call processing can lead to a better structured system with more consistent partitioning of call features.

In this paper, we describe an architecture for call processing and an implementation using object-oriented design techniques. The system is called *Archos*[2] and it is the controller for the XUNET II ATM switch, deployed in the XUNET II gigabit network testbed linking the University of California, Berkeley, the University of Wisconsin, Madison, the University of Illinois, Urbana-Champaign and AT&T Bell Laboratories, Murray Hill, N.J. It runs on the *Choices*[Rus90][CRJ87][RC89][CM90] object-oriented operating system, which is customizable and has been ported to a wide variety of machines. The object-oriented approach provides consistent partitioning of call features, easy maintenance and simple prototyping of new features. Because *Choices* is a multiprocessor operating system, parallel processing can scale performance of the switch controller as utilization grows. In the sections that follow, this paper:

- provides background
  - on the XUNET II switch,
  - and on the architectural models for call processing software structure, showing why one particular model (the *Process Per Channel* model) is superior.
- describes the software architecture of *Archos*, the use of *active objects* in the architecture, its supervision and signalling protocol (called GULP), the memory management of active objects, and finally gives details of version 2 of the implementation.

2 Background

2.1 The XUNET II Switch

The XUNET II switch is the basic switching component in the XUNET II Gigabit Network Testbed (Figure 1). The XUNET II switch is a re-engineered descendant of the Datakit[A. 89]. It incorporates concepts developed for broadband ISDN (Integrated Services Digital Network)[DC85], specifically Asynchronous Transfer Mode (ATM). Video, voice and data are broken into small, fixed size units called cells for transmission. Cells have a common format: a 5 byte header and an 48 byte payload. Cells from different conversations can be mixed together and switched uniformly. The switch is designed to handle transmission rates of up to 1 Gigabit per second.

2.1.1 Modules and Buses

Figure 2 shows a block schematic of its functional components. Four types of modules are shown in the diagram:

- line modules,

[2]The name *Archos* was coined by Ravi Sethi and Chuck Kalmenek of AT&T Bell Laboratories, Murray Hill, N.J. It has no relation to the real-time operating system ArchOS[Jen85].

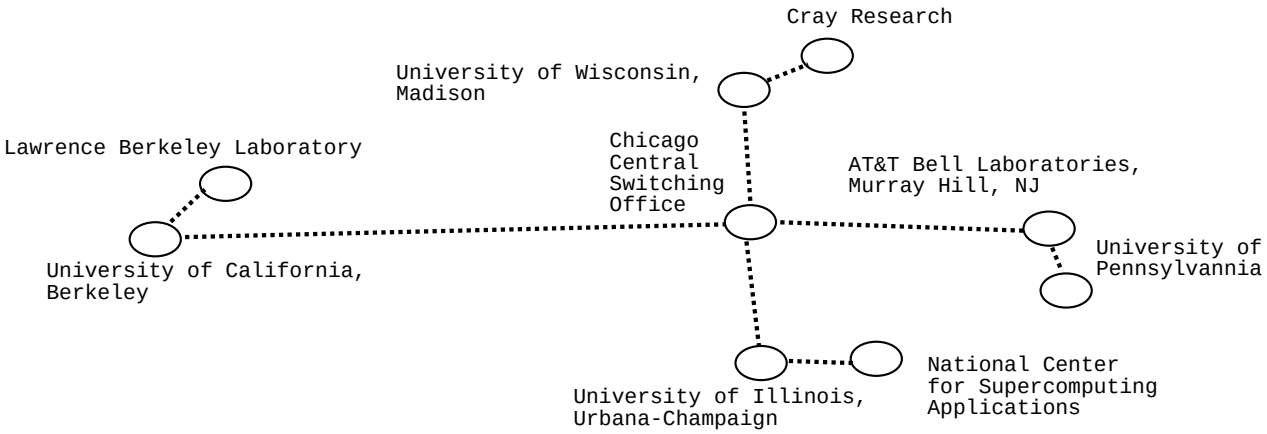

Figure 1: Map of the XUNET II gigabit network testbed.

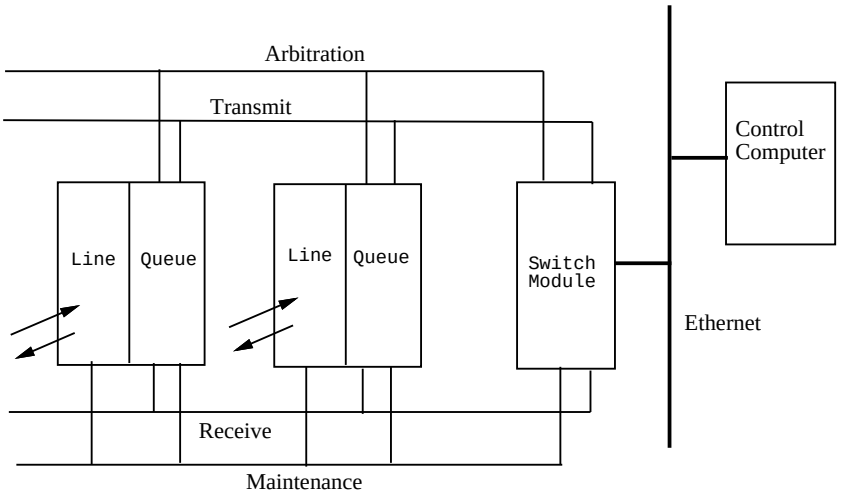

Figure 2: Block schematic of the XUNET II switch.

- queue modules,
- a switch module, and
- a control computer.

Line modules are the input/output interface units to trunk transmission lines. In XUNET II, they interface to 45 Mbit/s fiber access lines. Queue modules provide queueing for incoming cells for transmission to the switch module, and per channel queues for outgoing cells received from the switch module. These modules are cards set into slots in the switch. A queue card and a line card pair make up one trunk interface module.

XUNET II is also a bus-based architecture. Four main buses in the architecture comprise the switch backplane. They are the arbitration bus, transmit bus, receive bus and the maintenance bus. The buses provide a communication path between the trunk interface modules and the switch

module. For the XUNET II switch to be fast enough in switching packets, the backplane operates at five hundred megabits per second, about ten times the trunk rate. It is able to perform one million virtual circuit address translations per second.

When a cell arrives at a line module, it is placed in an input queue in the associated queue module. The arbitration bus resolves contention among the various queue modules for transmission of their incoming cells to the switch module according to a round robin scheme with two levels of priority. The algorithm results in all but one module withdrawing from the contention. The cell is transmitted to the switch module on the receive bus once the queue module wins the arbitration.

The switch module performs an address translation on the header of the cell. It translates the incoming pair <**incoming module number, incoming channel number**> to <**outgoing module number, outgoing channel number**> by performing a lookup in its internal lookup table. The translated cell is then sent out on the transmit bus and is received by the destination queue module, which queues it for outbound delivery on its line module. There are per channel queues for 64K virtual circuits in each queue module. The service discipline for each queue module is programmable.

The maintenance bus provides a communications path for sending status requests and reports between the various trunk interface modules and the switch module.

2.1.2 Switch Module

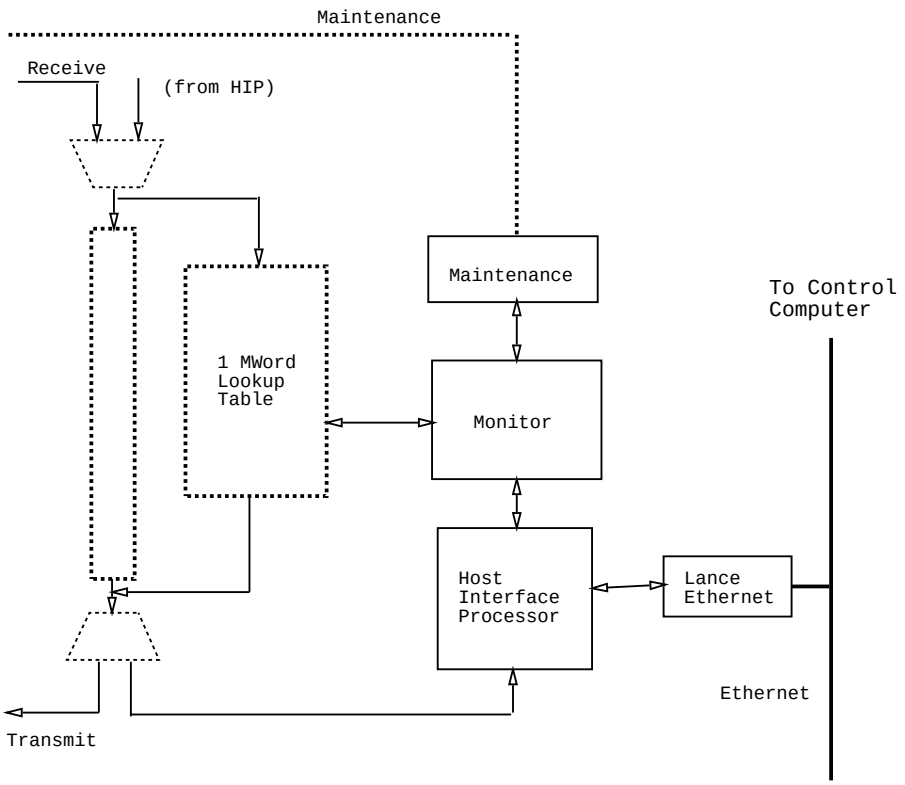

Figure 3: XUNET II switch module

Figure 3 shows a block schematic of the switch module. The monitor is a Motorola 68020 microprocessor (MC68020) running a monitor program for the switch. The XUNET II switch memory is made up of 64 bit words that are accessed in two 32 bit banks. The first bank is translation memory: a 5 bit destination slot number, 16 bit destination channel number and extra bookkeeping bits. The other bank is a cell count for the virtual circuit. Memory size is 1 MWord. This memory functions as a lookup table for performing virtual circuit identifier translations. It is mapped into the address space of the MC68020 microprocessor. The host interface processor is another Motorola 68020 microprocessor. It is dedicated to communicating with the control computer. It passes commands from the control computer to the monitor and sends responses to those commands back to the control computer.

The maintenance block is an EEPROM programmed microcontroller (87c751). Every module has a similar microcontroller that communicates with the maintenance block through the dedicated maintenance bus. The maintenance block is the master unit on the maintenance bus. It regularly polls for the state of the various trunk interface modules in the switch. The cards reply with their type, serial number and status information. The control computer can send commands to the maintenance block and set, enable or disable specific trunk interface modules.

2.1.3 Control Computer

The control computer is necessary for setting up and taking down virtual circuits. Note that the control computer is separate from the rest of the switch. It communicates with the switch across an Ethernet[3] cable to the switch module. The host interface processor relies on the Lance Ethernet chip set for transmitting and receiving on the Ethernet. The host interface processor accepts packets in either raw Ethernet format or UDP/IP[4][Pos80]. Responses from the monitor are sent back in the same format as what was received.

The control computer can therefore be any machine that is capable of exchanging data with the host interface processor on the Ethernet. This flexibility allows the control hardware, software and operating system to be independent of the switch.

Other hosts, including more than one controller, can reside on the same Ethernet with the switch.

2.2 Architectural Models for Call Processing

Figure 4 shows an abstract model of a circuit or virtual circuit-based network. The call processing models discussed below operate in such a framework.

It is assumed that communications in the network occur on virtual circuits. Data flows on virtual circuits between different *devices* in the network. Different types of devices can be connected to the switch. For example, Figure 4 shows two types of devices, a *host computer* and a *gateway*. A host computer can be a workstation or a mainframe. A gateway[Tan88] serves to route data from a different type of network to the virtual circuit network and vice versa. In XUNET II for example, the router moves data between a Fiber Distributed Data Interface (FDDI) based network and the XUNET II virtual circuit based network. *Device interfaces* allow the switch software

[3] Ethernet is a trademark of XEROX Corporation

[4] User Datagram Protocol/Internet Protocol, a set of protocols specified by the Department of Defense for computer communications.

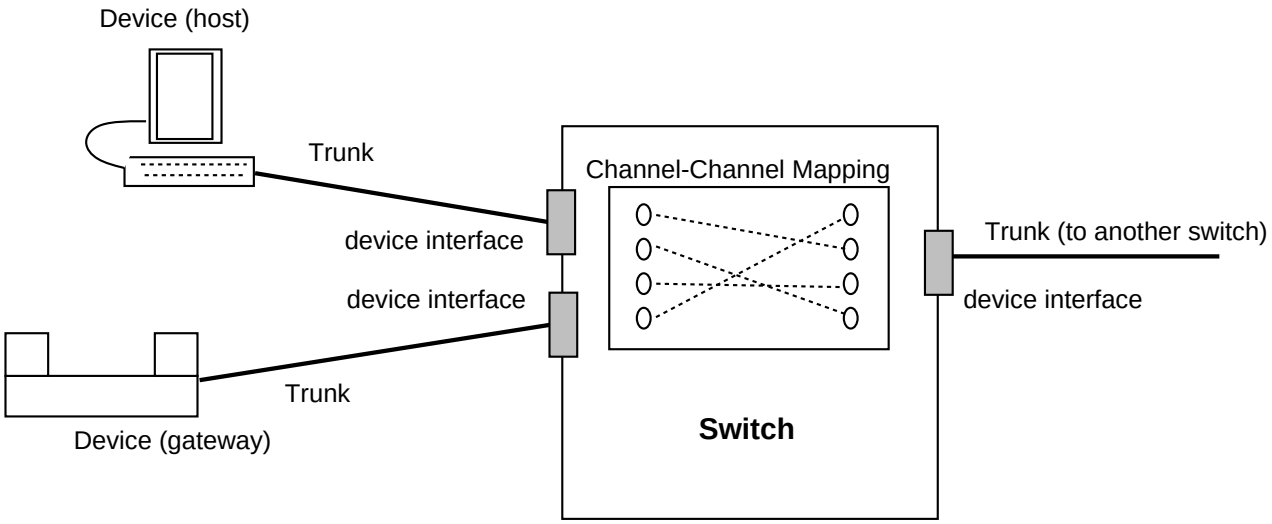

Figure 4: Components in the network.

to communicate with the device. It is likely that different devices will require different device interfaces.

Trunks connect devices to the switch. A trunk can have many *logical channels*. Each channel has a unique *channel number*, usually in the range 1 to some maximum number. Each channel is a duplex link for data between the source and destination. Notice that a trunk can connect a switch to another switch. A switch can thus also be considered a device.

The switch has a *channel-channel map*. The map is used to translate data coming from one logical channel on a trunk to another logical channel. Control software runs in the switch to maintain the settings in the map. This software is called *call processing software*. Two channels are *connected* by setting the channel-channel map such that data is forwarded from one channel to the other.

A network is a graph that is formed by taking the devices as vertices and the trunks as edges in the graph. A *virtual circuit* is a sequence of connected channels through the graph terminating at two devices. The *route* is the sequence of trunks used for the virtual circuit.

A virtual circuit is set up by devices exchanging *control messages*[5]. Call processing software exists to set up a virtual circuit by manipulating the channel-channel maps in the switches on the route, so that channels on the route are connected together. After the circuit has been set up, data can flow on the virtual circuit. Control messages and data messages are disjoint. That is, one must be able to distinguish a control message from a data message. This is usually accomplished by having control messages all use a special reserved channel. For terminals connected to a switch with single channel trunks (lines), control messages often take the form of special signals from the terminal to the device interface and vice versa.

[5] In telephony, control messages are also often termed *signals*.

2.2.1 Models for Call Processing

A single process solution to call processing is not viable, since many events occur asynchronously in a switch. For example, call setup requests and answers to those requests arrive asynchronously, and calls wait for responses. With many channels, there are many asynchronous input streams. The state machine constructed with a uniprocess model would be enormously complex and unwieldy. In designing the TDK system for the Datakit, McMahon notes:

> "The job of controlling a switch is a parade example of a multithread job. The computing required to connect two consenting lines[6] is trivial; the only hard part of the controller is to manage many connections in various states of completion at once." [McM81]

Instead, responsibilities should be partitioned among multiple cooperating processes in the system. The manner of assigning processes to tasks is the clearest distinguishing feature of a call processing software architecture. In this section, two basic models for call processing are examined, namely the **Process Per Phase** and the **Process Per Channel** models. Both models are abstract, each is orthogonal to the other, and all existing call processing systems lie within the spectrum of design choices that the two models span.

2.2.2 Process Per Phase Model

Consider a typical telephone call – the call will move through several distinct phases:

1. The state of the line connecting the telephone to the switch is first idle (idle phase).
2. A connection request arrives at the switch (offhook phase). A dial tone must be applied so that the user hears the tone.
3. Dial pulses arrive (signaling phase).
4. The call is dialed across the network (dialing phase).
5. On the receiving end, a ringing tone is applied to the destination telephone (ringing phase), and the corresponding ringing tone is relayed to the orginator.
6. If instead, the destination is busy, a busy tone is returned to the caller (busy phase). The call returns to the idle phase when the receiver is set down.
7. If the destination line is not busy, then the two ends are connected if the destination telephone is picked up (talking phase).
8. The call ends when one side decides to disconnect (disconnect phase). The disconnect message is relayed to the other side of the call, then the line will enter the idle phase again.
9. The other side gets a hangup signal (hangup phase). The call enters the idle phase again once the receiver is set down.

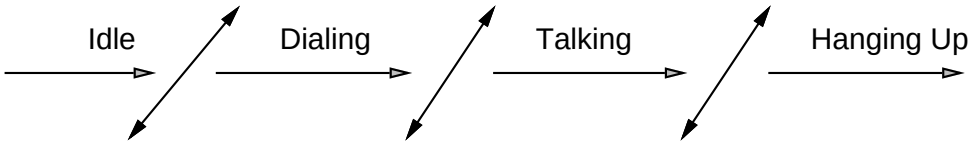

Figure 5: Dissecting a call into phases.

The Process Per Phase model assigns a process to handle each phase. Thus, in the POTS scenario sketched out above, the software system will have 9 processes representing the different phases[7]. Calls transit between between the different processes as they go through their life cycle.

AT&T's No. 1ESS[CDCH] and the Italian Italtel UT 10/3[Ita] use this model.

2.2.3 Process Per Channel Model

In the Process Per Channel model, a call is associated with a logical channel. The Process Per Channel model [McM81] assigns a process to manage the state of each channel, thus, every channel has an associated process. Such a process can be specified as a finite state machine (Figure 7).

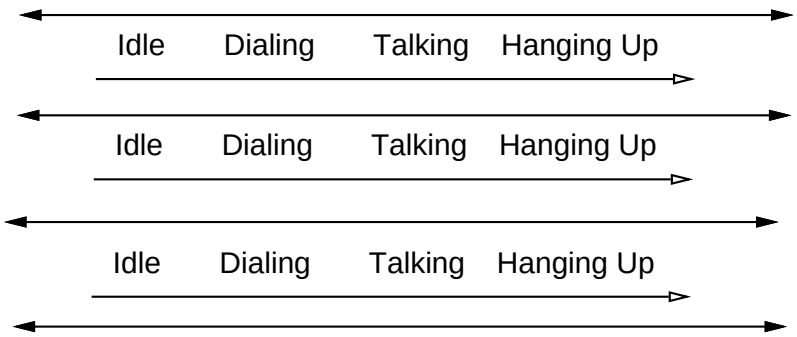

Figure 6: Dissecting by separating calls from each other.

The TDK[McM81][Che79] system for the Datakit introduced the Process Per Channel model. Because TDK was originally designed for terminals connected to switches through lines (single channel trunks), the model is called Process Per Line in [McM81] and [Che79]. [MS90] describes an implementation on the TDK system of terminal call processing with Esterel, a language specialized for handling finite state machines.

2.2.4 Comparison

The Process Per Phase model "cuts across the grain" of the lifetime of all calls by isolating calls in their phases. In contrast, the Process Per Channel model "cuts along the grain" by separating calls from each other. In this sense, they are orthogonal models.

The Process Per Phase model is simpler when the number of phases is small, such as in a two phase case, where a channel is either connected or not-connected. To set up calls, the process

[6] TDK was originally designed to support terminals connected to a switch. McMahon thus speaks of connecting *lines* instead of channels. In the model, a *line* is a trunk with a single duplex channel.

[7] A corresponding set of 9 signals are defined in the United States standard [ATT80, Sch87].

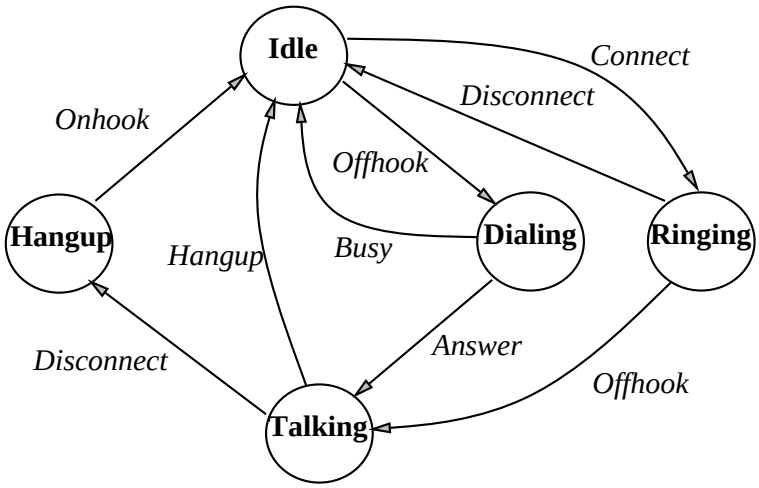

Figure 7: Simplified finite state machine for a POTS call.

governing the not-connected phase receives requests indicating which channels to set up for calls. Those channels which become connected then pass to the charge of the other process governing the connected channels. A symmetric relationship occurs for call takedown. With the Process Per Channel model, a large number of channels will result in an equivalent number of processes. The Process Per Phase model is simpler in this case.

However, it is likely that the number of distinct phases that the software must handle will be greater than two. In the earlier analysis, nine phases were identified for a simple POTS[8] call. A larger number of phases forces each phase to become more complex, as there are more interactions between phases.

To allow different devices and their device interfaces to communicate with the switch, the controller needs to support all the peculiarities of the devices. In the Process Per Channel case, different process types can handle different devices. The knowledge a channel process has of its own device is encapsulated within the process and is not spread throughout the whole system. In the Process Per Phase case, each phase must handle each type of device. Modifications have to be made to every phase for each new device introduced. With a large number of device types, each phase becomes very complicated. The Process Per Phase model does not scale well if the number of devices types supported is increased.

Code for a different process type can be added to a Process Per Channel model while the system is still running. Dynamic linking[DSS90][MCK91] adds new code to a running program. This allows a new process type to be added to already running code. Support for new devices or new features can be introduced without any system down time. The Process Per Phase model has no such ability; adding a new process type will need reloading the programs for all phases. Adding devices without bringing a switch down is an important consideration in a production network.

There is one disadvantage to the Process Per Channel model: the numerous objects and processes needed to manage every channel. However, note that an implementation with proper memory management of these processes need only maintain a cache of active channel processes in memory at any moment. A working set[Den68] of these processes can ensure that an acceptable call setup

[8] Acronym for "Plain Old Telephone Service."

or takedown rate is maintained without keeping every channel process in memory. *Archos* draws from the Process Per Channel concept. Chapter 5 will explore how *Archos* manages the memory requirements and caching of its collection of channel processes.

3 *Archos* Software Architecture

The *Archos* software architecture takes its inspiration from Lee McMahon's work on TDK and TDK2[McM81, McM83]. *Archos* extends the previous work by fully exploiting the Process Per Channel model. Indeed, *Archos* is better described as an *Active Object Per Channel* model for call processing. Drawing from the *Choices* operating system, *Archos* is also a *framework*[Mey87][JR91] for constructing call processing systems[9]. The classes in *Archos* make up a family of call processing systems that are specialized for various features and can work on different switches.

The following sections detail the design of *Archos*.

3.1 Object-Oriented Modeling

In [McM81], Lee McMahon notes of the Process Per Channel model:

> "The main advantages of this organization are ... the idiosyncracies of a single kind of line are concentrated in the process controlling that kind of line."

Different types of devices require different kinds of handling. The kinds of signals each device expects are different. However, some main steps in the call processing procedure for all types of devices hardly differ. For example, these four steps in the procedure are needed in all devices:

1. sending a call setup message to make a call,
2. acknowledging receipt of a call,
3. negative acknowledgement, meaning the call is refused,
4. hanging up.

A dumb terminal with a RS-232-C interface might signify readiness to set up a call by asserting its Data Terminal Ready lead, while a workstation might send a special control message.

Every channel is managed by a process in the Process Per Channel model. Channels belonging to different devices require different handling, and therefore different process types. A class hierarchy of process types can be created to provide a framework for channel processes that factor out common behaviour, and the common code in the superclass will contain methods for the four steps above. Subclasses will implement concretely the protocols required for particular devices. If a group of devices are similar in many ways, a superclass can factor out their common features. The strengths of object-oriented programming in call processing is subclassing applied toward the problem of

> "this device works almost like this other one, except for ..."

[9] The *Choices* operating system is constructed using object-oriented techniques. Abstract classes capture important machine-independent algorithms and policies, and determine the interactions between the components in the system. Subclasses concretely implement the details that are needed for particular policies and machine-specific requirements. The classes provide a *framework* upon which concrete systems can be built by "mixing and matching" components.

Process types in *Archos* are represented as objects of an abstract class Channel. Subclassing provides concrete implementations for the different devices supported in the system. The Process Per Channel model is retained in *Archos* by making Channel an *Active Object*.

3.2 Active Objects

An active object in the *Archos* system is an object with a thread of execution associated with it. Each thread is independently schedulable and runs within the context of the object. The object provides a repository of state information about the channel it is managing. In addition, it is autonomous and can react independently to events. Active objects communicate by passing messages to one another. Each channel on a trunk is managed by two active objects running in

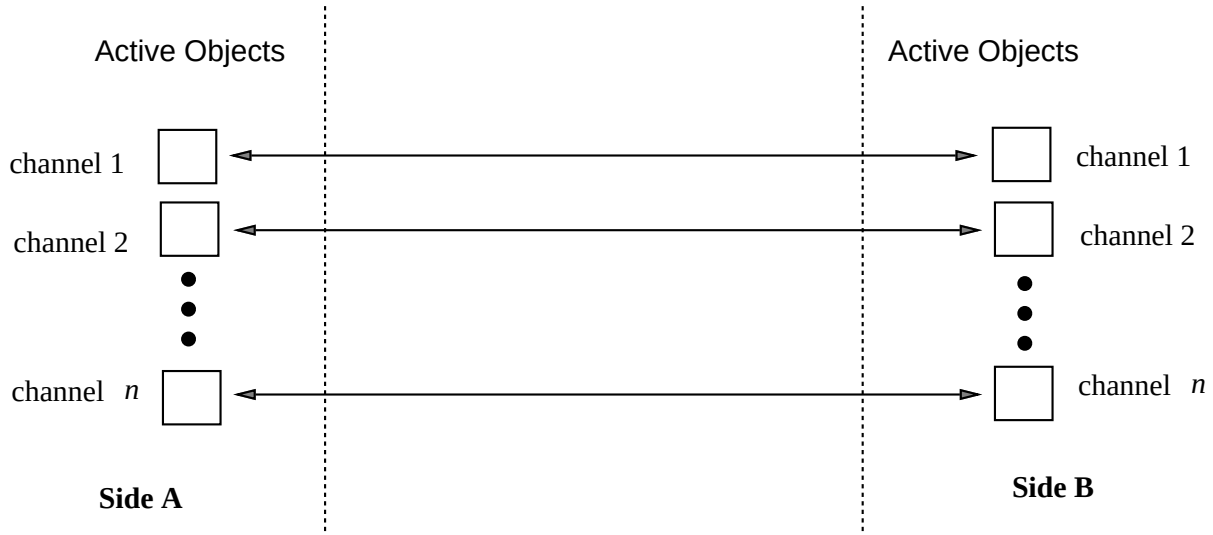

Figure 8: Every active object has a peer.

the control computers at each end (figure 8). The peer active objects cooperate to manage a single channel. Across the trunk, an active object communicates solely with its peer, although other active objects on its own side can communicate with it. Crosstalk between non-peer active objects across a trunk does not happen.

3.3 Common Supervision and Trunk Control

One channel called the *common supervision channel* is always open on a trunk between two nodes in the network. All active objects for a trunk share that channel for exchanging messages with each other[10]. The trunk control manages messages for its collection of active objects. A active object wishing to communicate with its peer passes its message to the trunk control. The trunk control sends the message out on the common supervision channel. Messages are received at the other end by its trunk control and passed on to the recipient object. Trunk control handles the multiplexing and demultiplexing of messages to active objects on the common supervision channel.

[10]Every channel is governed by a pair of active objects at each end, but if every pair of active objects had a channel dedicated for exchanging messages with each other, then there would not be any circuits left over for calls.

4 Supervision and Signalling Protocol

Communication between active objects is message based. The protocol is called GULP — Generic Universal Line Protocol. GULP messages sent to the trunk object are forwarded across the common supervision channel to the other side and are demultiplexed at the receiver.

Message	Meaning
V_START	Active object ready to start
V_CALL	Call from within switch/host
P_INCALL	Call from outside switch/host
V_ANS	Answer to V_CALL or P_INCALL
V_NAK	Negative answer to V_CALL or P_INCALL
V_HUP	Hangup message for a call
V_HUP_ACK	Hangup acknowledgement

Table 1: GULP messages and their meanings

GULP is used for synchronization between peer objects managing a channel, and for passing call setup and takedown messages. It is the only signalling protocol used within an *Archos* system. The messages are listed in table 1[11]. This section describes the section in more detail.

4.1 Synchronization

Active objects are created at system start up time. They synchronize with their peers before processing any calls. The synchronization procedure is as follows: one side sends a V_START message to its peer and waits for a V_START from its peer; when both sides have received V_START messages, they are then synchronized.

Active objects resynchronize with their peers after every call setup–call takedown cycle. Any errors also result in resynchronization.

4.2 Calling

The GULP calling sequence is shown in Figure 9. A V_CALL message is sent to an active object from another active object within the computer when the originator wants to make a call out. Both V_CALL and P_INCALL messages carry an implicit dialstring. On receipt of the V_CALL, the active object sends a P_INCALL message to its peer, and waits for a response.

P_INCALL messages are received from an active object's peer on the other end of the trunk. On receipt of a P_INCALL message, an active object looks up the destination by performing a *translate* function on the dialstring. The function should return a destination active object[12]. If no active object is returned, then either the dialstring is invalid or all channels to that destination are busy, and the call fails. A V_NAK is returned to the originating active object. If a non-busy destination

[11] GULP is a unified supervision and signalling protocol descended from TDK's *Virtual Line Protocol*. Hence most of the GULP messages are prefixed by Vs. The one exception, P_INCALL, should be read as "peer incall", and is thus prefixed with a P.

[12] TDK uses static routing tables with a hierarchical naming convention For example, "il/uiuc/cs/ligarius" names the host ligarius in the Computer Science Department of the University of Illinois, Urbana-Champaign, in the state of Illinois. *Archos* currently uses TDK's static routing scheme.

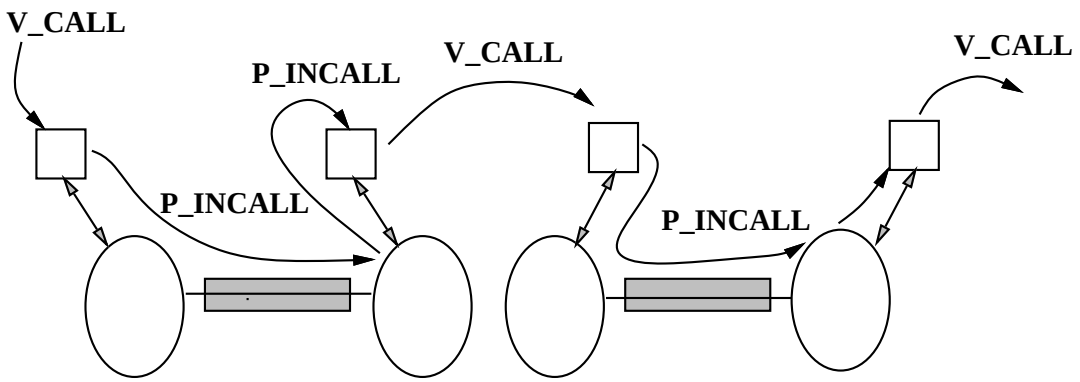

Figure 9: GULP calling sequence

active object is indeed found, a V_CALL message is sent to the destination, which propagates the call onto the next switch in a similar fashion until an end-point is reached.

If the receiver of the V_CALL is an end-point, it sends back a V_NAK if the call cannot be completed, or a V_ANS if the call can be. Both replies are propagated back to the originator.

4.3 Hanging Up

Either end of the connection is free to issue a V_HUP message to hang up the connection. The V_HUP is propagated from active object to active object, across trunks to the other side, where a V_HUP_ACK is sent back. After an object forwards the V_HUP_ACK, it resynchronizes with its peer.

4.4 Dialstring Propagation

GULP messages V_CALL and P_INCALL carry destination dialstrings with them. Dialstrings are therefore carried along on the Common Supervision Channel between devices. In contrast, TDK's Virtual Line Protocol, VLP, requires that dialstrings be carried on data channels (Figure 10). A call setup in TDK proceeds with the source machine sending a VLP message to its switch requesting an outgoing channel. The switch software allocates a channel to the source, maps it to a line process in the switch software, and then informs the source over the the Common Supervision Channel of the channel number allocated. The source then opens a connection on that channel and sends the dialstring over to the line process on the data channel. The line process translates the dialstring and decides the route to take. Setup negotiations akin to the above procedure take place between each pair of nodes in the path. Success or failure is returned to the source on the data channel.

Dialstring propagation is more cumbersome in VLP than in GULP. By unifying the transport of supervision (protocol messages) and signalling (dialstrings) onto one channel, there is no need for separate data channels for dialstring information. In GULP, the call setup message can propagate without the overhead of first setting the switch map to read the dialstring, and then later remapping the data channel to an outgoing channel when the call is successful. Minimizing the number of writes to switch memory for call setup speeds the procedure up[13].

[13] In measurements of TDK performance, writing switch memory accounted for up to 75% of the time required for

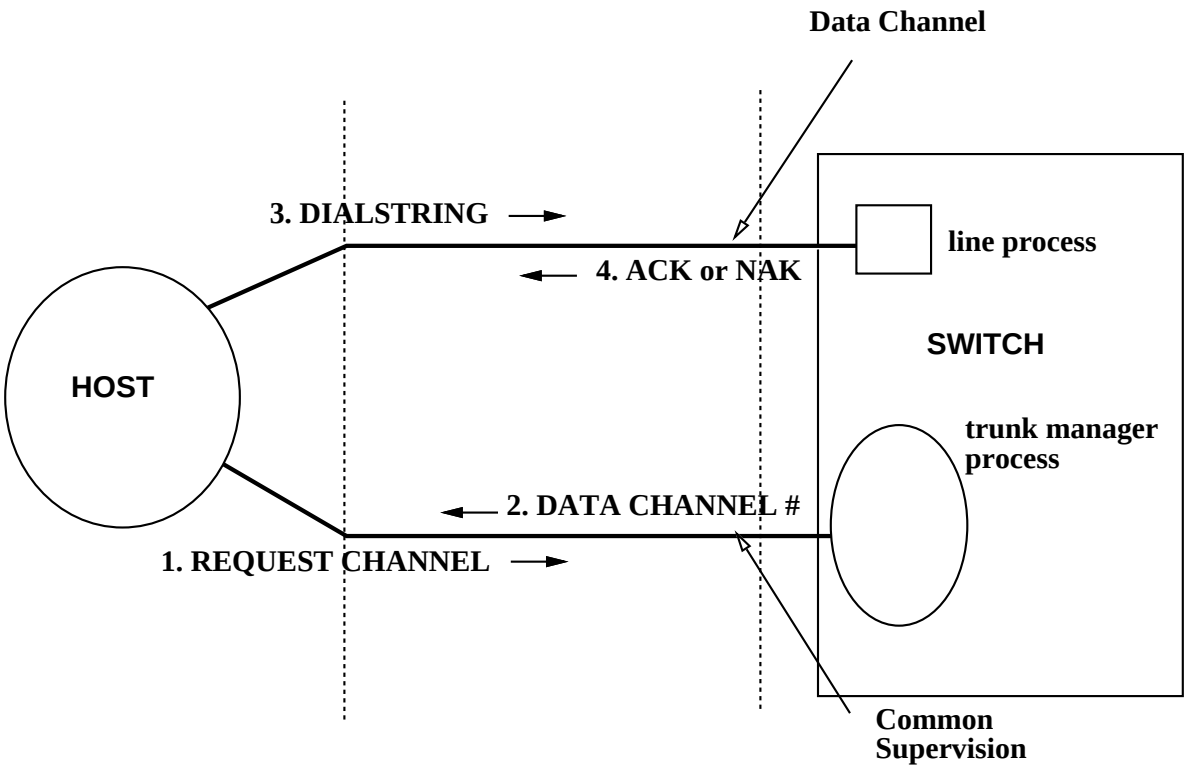

Figure 10: TDK dialstring propagation during call setup.

In the *Archos* implementation, anticipatory setup of the channel-channel map for calls is done in parallel with the propagation of call setup information. That is, the switch's channel-channel map is set to translate traffic from the incoming channel to the outgoing channel (and vice versa) in anticipation that the call will succeed. This occurs in parallel with the setup messages propagating over the network. If the call is indeed successful, then no further writes to switch memory is required when V_ANS messages are received. Call setup is thus sped up.

4.5 Channel Contention

Channel contention occurs when both peer active objects wish to initiate calls at the same time. One side may send a P_INCALL to its peer and then receive a P_INCALL from the same peer. Both sides wish to place a call on the same channel.

Contention resolution can be based on predetermined priority: the side with the lower priority always backs off and allows the other side (with the higher priority) to continue with its call. In this case, priority can be predetermined by, for example, both sides exchanging random numbers at system start up time.

The case for channel contention can be limited to the last channel used if GULP is augmented with a High-Low channel allocation scheme.

This scheme uses the convention that the low priority side allocates channels from low to high,

call setup.

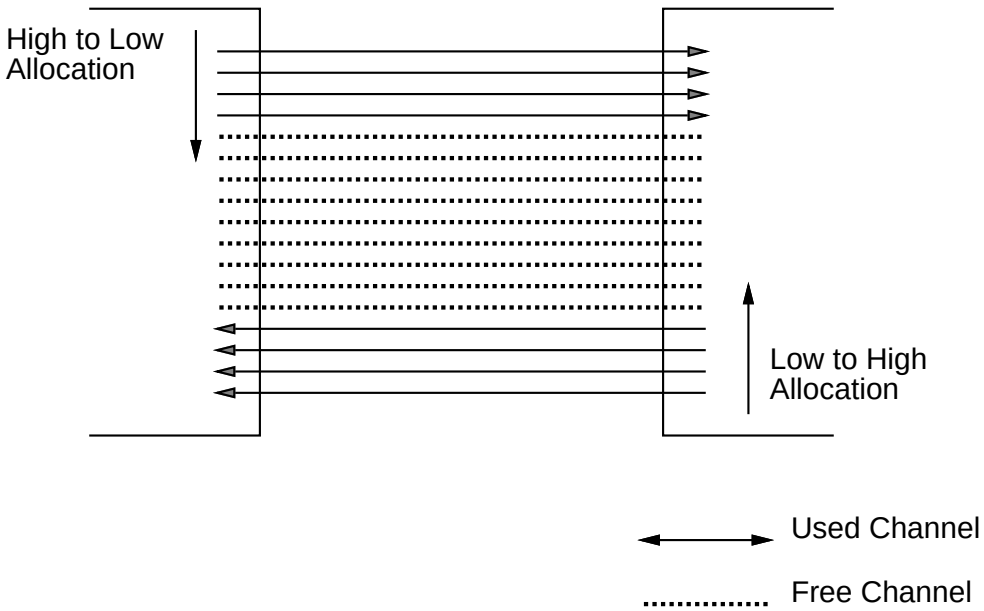

Figure 11: High-Low allocation scheme

and the high priority side allocates channels from high to low. In this way, no contention will occur until all channels but the last one have been allocated. Contention occurs for the last channel if both sides wish to place calls simultaneously. The lower priority side withdraws its call setup attempt and allows the higher priority side to continue.

Even though one side is always inferior, the High-Low scheme limits contention occurrence to the last channel used. Note that call setup oscillations can occur in channel contention resolution if priority is used. A high priority peer can win a channel to a certain point in a virtual circuit as it is being set up, only to have the channel torn down when a low priority peer further along the route loses. This imposes an extra workload on the network. Contention-caused oscillation is more likely if many channels in the network are used up.

Channel contention can be eliminated entirely if the last channel is never allocated. Although equivalent to both sides losing the contention for the last channel, it will mitigate oscillation problems if many channels are used up in the network.

5 Memory Management of Active Objects

Since *Archos* uses the Active Object Per Channel model, it can have a large number of active objects. Each active object requires memory for its data. A process also requires a stack to hold member function activation records. These needs require a large amount of physical memory if all objects and processes are kept in memory — memory that might not be sufficient on the control computer.

The operating system needs an efficient way of managing the active objects in the system. Unused active objects can be swapped out, their object data and stacks removed to secondary storage. Only a cache of currently active objects must remain in memory.

Most operating systems do memory management by removing infrequently used pages of memory to disk. However, thrashing[PS85] can result if data is paged out only to be required very soon after the page out occurs. The operating system spends all its time paging and no useful work gets done.

Various paging policies are already implemented to reduce the chance of thrashing. The theoretical best algorithm is the MIN or OPT algorithm. The algorithm directs the system to remove the page that will be used furthest away in the future. Extending this to *Archos*, the system should remove the channel process object which will be used furthest in the future.

This is not a realizable algorithm, since it is impossible to predict which page or channel process object this will be. However, other algorithms such as LRU — Least Recently Used — give good approximations to MIN. The LRU paging algorithm replaces the least recently used page. LRU is based on the *principle of locality*[PS85]. There are two cases for locality: if a program accesses a particular page in memory,

1. *temporal locality* surmises that it is likely to access it again soon, and

2. *spatial locality* surmises that data around that area will also be acccessed again soon.

If a page has not been accessed recently, then the focus of the program has shifted to another locality, and thus it should be safe to remove. It was however, unclear if the least recently used call object would necessarily be the one most likely to be used furthest in the future.

A study of call traffic on a Datakit based network was carried out to discover if any patterns existed that would relate to channel object memory management. Those results and conclusions are presented below.

5.1 Traffic Measurement

Seven Datakit switches serve the Computing Research Center of AT&T Bell Laboratories at Murray Hill, N.J. Approximately 50 researchers plus some clerical staff use the switches in a local area network in a research and development environment. Most of the machines at the Center run either Tenth Edition UNIX[14] [RT75] or the Plan 9 operating system[PPTT90]. Typical use of the network includes remote login, file transfers and connections made by daemon processes[15].

Logs of Datakit activity were collected for twenty-four hours of normal operation. The information contained call setups and takedowns between source and destination channels and the time at which they occurred, with a 1 second granularity. Nearly fifty thousand actions[16] were logged.

In the following sections, an *interval* is defined as the idle period delimited by any activity on a channel. Therefore, a setup action to a takedown action counts as an interval, and so does the period between takedown and the initiation of another action. An interval is *killed* by an action occurring on that channel. This new action then defines the start of another interval.

[14] UNIX is trademark of Unix Laboratories, Inc. Tenth Edition UNIX is a research UNIX not used outside of the Center.

[15] In UNIX slang, daemons are background processes which run and provide operating system services, for example, mail service.

[16] An action is either a connect, connection acknowlegement, negative acknowledgement or disconnect message sent to a channel.

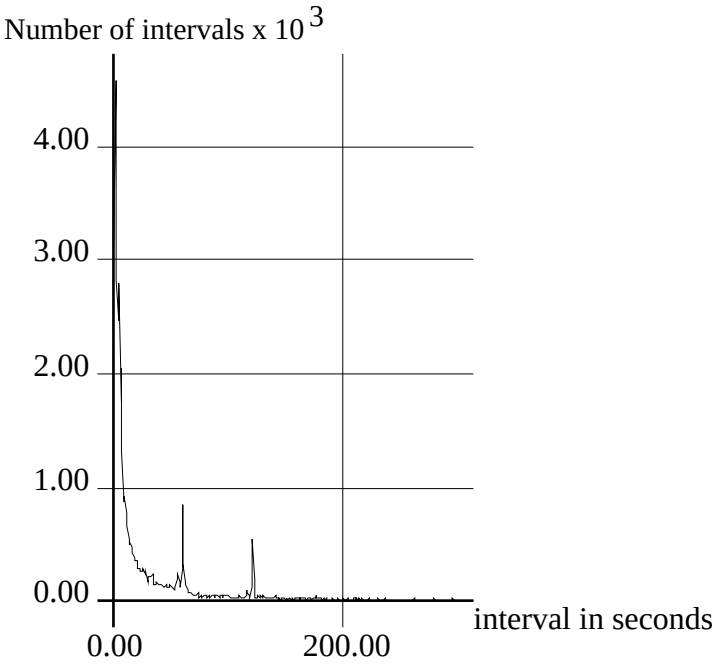

Figure 12: Number of intervals versus length of interval.

Time Between Activity Figure 12 shows a plot of the number of intervals versus the length of an interval in seconds, based on all the channels that were active on the Datakit switch. The longest interval was 81212 seconds. Short intervals dominated long intervals. Note that spikes occur at the 60 and 120 second marks. These were was caused by the periodic behavior of some system processes which make connections every minute or two minutes on the same channel.

Percentage Completion Figure 13 plots the percentage of intervals that complete within a given time period. Half of all intervals were killed within 10 seconds.

Expected Lifetimes Figure 14 plots the expected lifetime of an interval versus the time it has already been alive. Figure 15 plots the expected number of seconds more to live versus the time it has already been alive. That is, the figures show the expected number of seconds a channel will remain idle if it has already been idle for an given amount of time.

5.2 Discussion

To summarize the findings, Figure 12 and Figure 13 show that short intervals predominate. That is, most channels have short idle periods. Channel activity thus shows strong temporal locality.

Figure 14 and Figure 15 show that the longer a channel remains idle, the more likely it is to remain idle. Note that in the initial part of the Figure 15, from 0 to 10 seconds, the slope is extremely steep. For each additional second the channel is idle, it gains 50 seconds more in expected idle time.

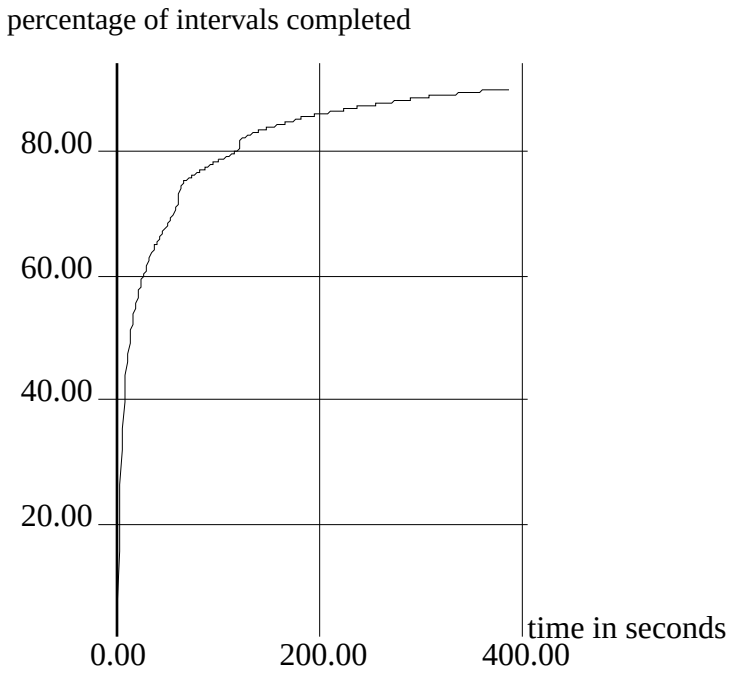

Figure 13: Percentage of intervals below a length of time.

Archos has one active object for every channel. There is therefore a direct correspondence between activity on a channel and on its active object. The longer an active object remains idle, the more likely it will remain idle. The LRU algorithm is directly applicable to virtual memory management of active objects. Indeed, with the expected lifetime gaining 50 seconds for every second alive, the LRU algorithm can be expected to give excellent results in the virtual memory management of call processing active objects.

The page-in time for an object depends on disk latency. For the AT&T 6386E WGS machine that the *Archos* prototype runs on, the *Choices* operating system requires at least thirty milliseconds to read a disk block into memory. The time needed to page-in a single object is close to the round-trip latency experienced by speed-of-light signals propagating across the country. Requiring an object page-in during call setup will markedly affect call setup time. Therefore, during call setup, *Archos* avoids retrieving objects from disk by prefetching active objects. In the cache of active objects, the system maintains, for each trunk, a number of active objects for channels that will be allocated next on new call setup requests. Call setups will therefore use an object already in the cache without needing to fetch one from the disk. As objects are used up, the system pages in more unused objects from the disk to maintain its prefetched cache. The size of the cache depends on the actual traffic load placed on the system.

6 *Archos* Implementation

The *Archos* implementation is distilled from the preceding discussion of the various aspects of call processing. *Archos* has had 3 rewrites, versions 0.1, 0.2 and the latest incarnation being version 2. Version 0.1 ran entirely within the *Choices* kernel as a set of kernel processes. Version 0.2 migrated

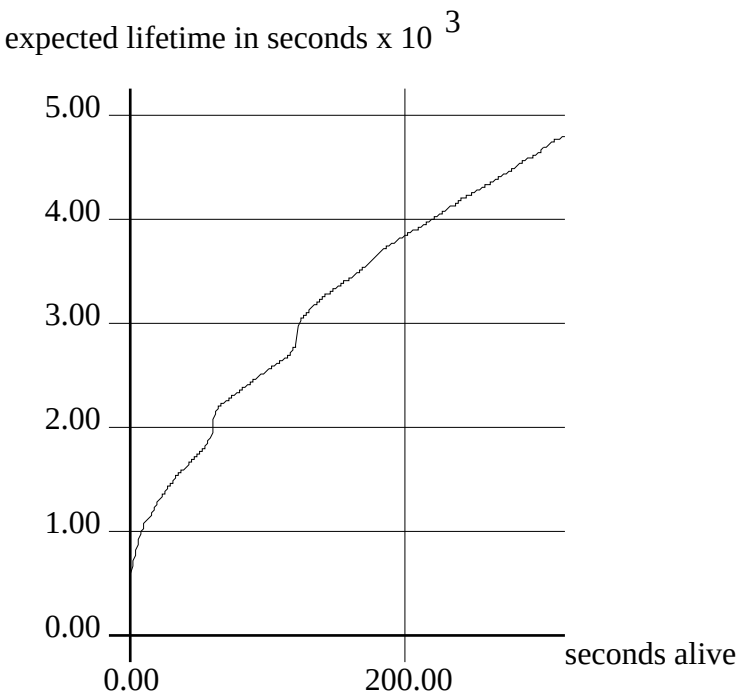

Figure 14: Expected lifetime of an interval versus its age

all the code to the user level and ran as a set of lightweight application level processes. After the data on calls placed in a Datakit switch were collected and analyzed (section 5), *Archos* was entirely rewritten to handle LRU swapping of active objects. The Datakit was used as the test switch in the two early versions. In version 2, an interface was added for the XUNET II switch. Because the change was fairly massive, the version number jumped 1.8 points.

The prototype is written in the C++ programming language on top of the *Choices* operating system. Using the object-oriented features of inheritance and code reuse, the code required for controlling a switch and the code that resides in a host end point connected to the network vary only in a few functions.

6.1 *Archos* Framework

All processes in *Archos* are lightweight, running within the same virtual address space, enabling access to common data structures with synchronization by shared semaphores. Lightweight processes reduce the cost of context switching between processes.

Archos is made up of three interlocking frameworks. They are:

1. the message passing system,
2. the channel objects, and
3. the channel manager system.

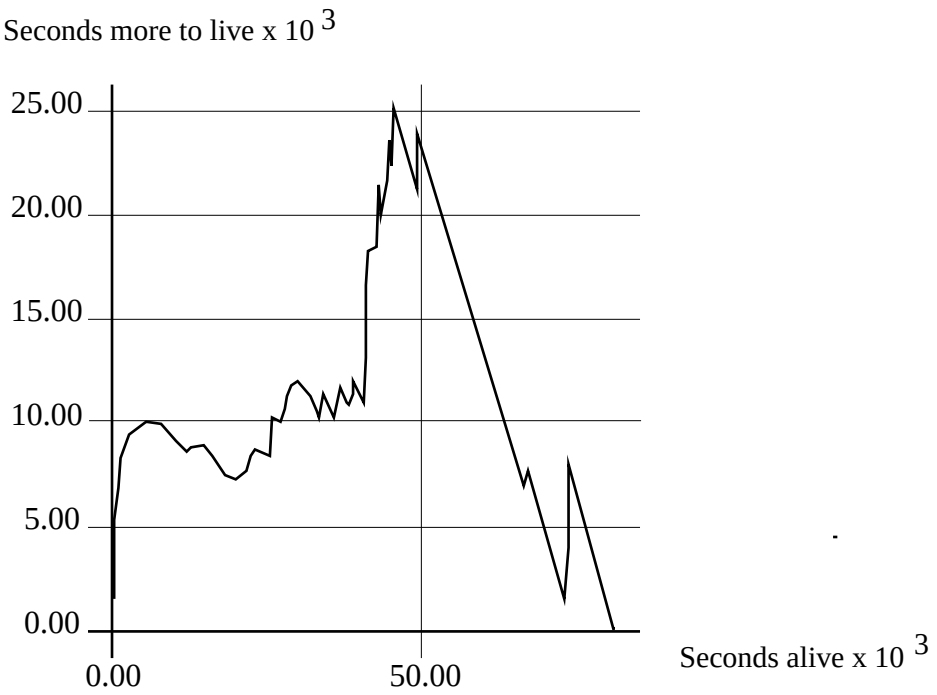

Figure 15: Expected time more to live of an interval versus its age

6.2 Message Passing System

The message passing system carries GULP messages between active objects. A channel object is denoted by its message address. A message address is the tuple <module number, channel number, message queue>. The message queue component of an address defines a "mail box" where the message should be placed for delivery. The MessageAddress class represents an address.

A message is a 5-tuple: <GULP code, sender address, destination address, associated data, length of associated data>. The Message class represents a message. The class has a member function deliver() that posts the message to the destination's message queue. Since the return address is always sent along, the recipient can always reply to the message.

The MessageQueue is a simple, semaphore synchronized queue that holds messages.

6.3 Channel Objects

The Channel class implements the GULP based call processing finite state machine for a single channel process (Figure 16). A Channel object contains its own message address, the message address of its peer, methods to send GULP messages and its current state.

Channel objects are reactive objects which act when a message is sent to them. Being a finite state machine, an object's action on receiving a message will end with the object entering a new state. While they are waiting for input, Channel objects do not need active threads of control executing within them. Thus, there is no need to store the entire stack of the process at the point where it is to wait for input. Only the new state of the object and other auxillary data (such as dialstrings, etc.) must be saved. This approach economizes on all the memory space required for

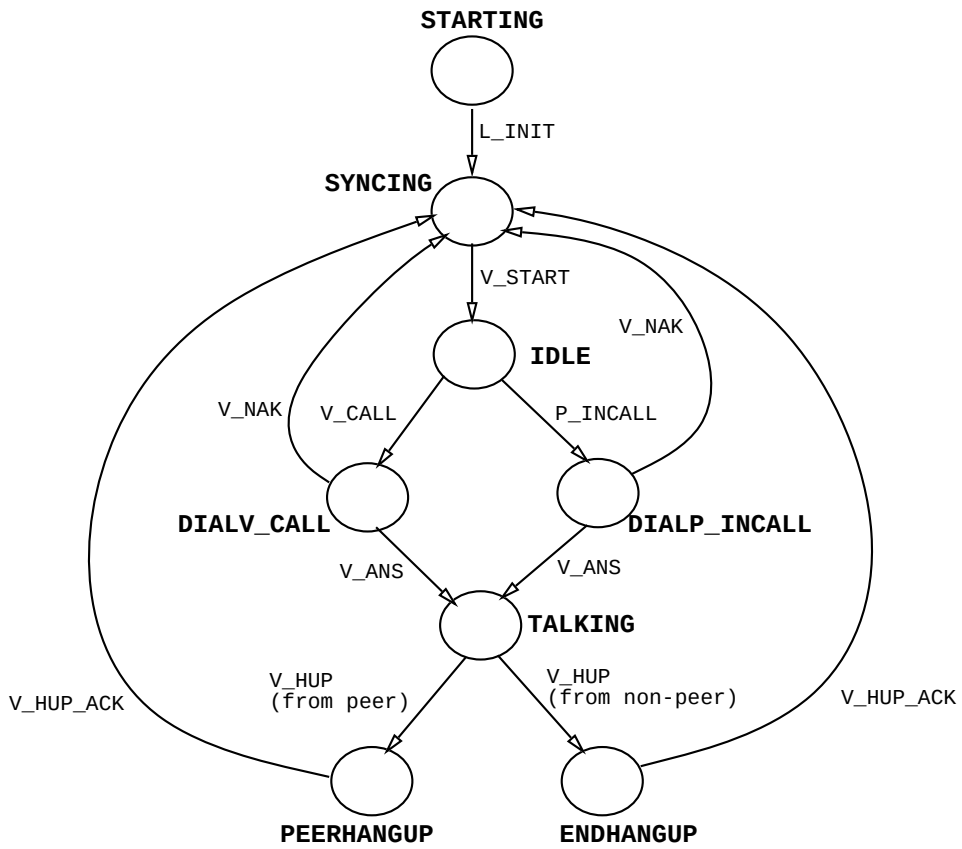

Figure 16: Simplified Channel object GULP finite state automata

processes, such as stacks and other kernel data structures, while still retaining the Process/Active Object Per Channel model.

6.4 Channel Manager

Instances of the ChannelManager class are containers of Channel objects for a trunk or a portion of the Channels of a trunk. For example, given a trunk with 64 channels, ChannelManager A could manage channels 1 to 40, and ChannelManager B could manage channels 41 to 64. ChannelManager A thus has 40 Channel objects under its charge, while ChannelManager B has 24. A ChannelManager can be thought of as a server of Channels. Messages arrive for Channel objects in the ChannelManager's message queue. A process running within the ChannelManager object, called the MessageReaderProcess, takes messages off this queue and looks up their intended recipient Channels in the ChannelManager's cache of active Channels. An attempt to access a Channel object not in the cache will cause the repairFault() method to be called. The required object is read back into memory from secondary storage. If the cache is full, the least recently used Channel object in the cache is removed and written to secondary storage.

Each Channel object in the cache is kept in a ChannelCacheUnit. A ChannelCacheUnit has a process that waits for incoming messages from its ChannelManager. The ChannelManager awakens

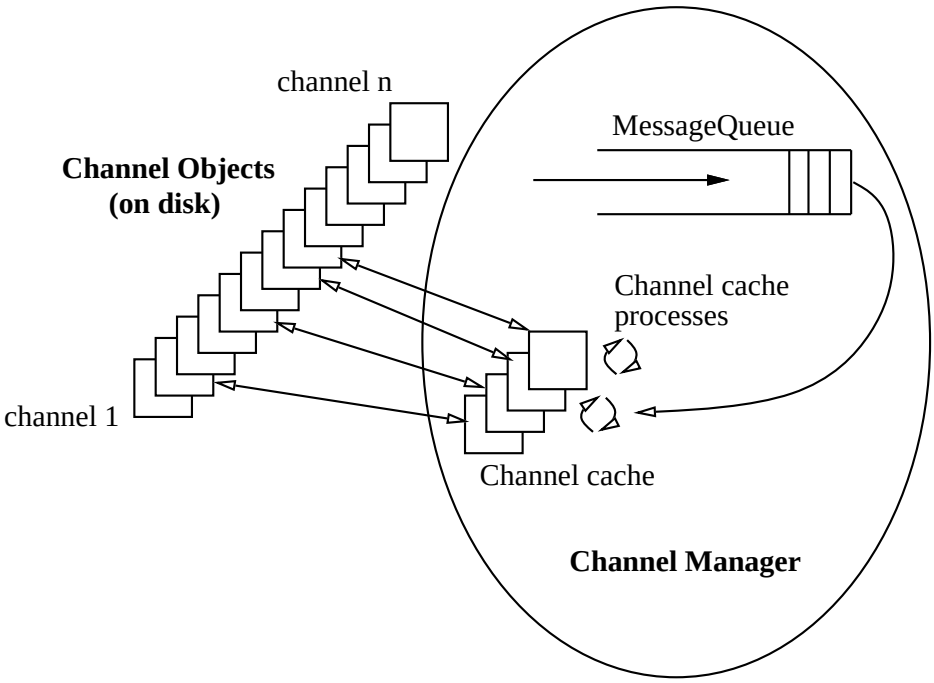

Figure 17: The Channel Manager

the process with the message. On receiving the message, the process starts executing within the ChannelObject. The TrunkMessageReaderProcess meanwhile goes on to service the next message on the message queue.

To prevent call setups from blocking while a required channel object is read back into memory from secondary storage, the channel manager maintains a number of channel objects for channels that will be allocated next on new setup requests. Call setups will therefore use objects already in the cache and will not wait for the required object to be paged in.

6.5 Specializing Channel for Switches and Hosts

Subclasses SwitchChannel and HostChannel of the superclass Channel handle the tasks of managing a channel on the switch side and on the host side respectively.

SwitchChannel The Channel object for a switch controller must manage, beyond acting as a finite state automaton for GULP, the setting of the switch's internal tables that connect two channels together. The SwitchInterface class provides an abstact interface to the switch. Its basic functions are:

1. connect(channel1, channel2), which establishes a connection between the two channels, and
2. ground(channel), which "zeroes" and disconnects one channel.

Xunet1SwitchInterface and Xunet2SwitchInterface are concrete classes for controlling a Datakit and a XUNET II switch respectively.

The SwitchChannel class redefines the member function in its superclass for the DIALV_CALL state (see figure 16). If a V_ANS message is received, which means that the call is accepted, a SwitchChannel object connects the two end points of the conversation together using the SwitchInterface's connect(...) member function. When the conversation ends, it disconnects the two end points with a ground(...) call to the SwitchInterface.

HostChannel The HostChannel class behaves exactly as its superclass Channel, except when a P_INCALL message is received. If the channel is not free, then the call is rejected by the HostChannel object for that channel. If the channel is free, then the HostChannel object may, for example, attempt to spawn a "login" process and accept the call.

6.6 Summary

Version 2 of the *Archos* implementation framework has 3 interlocking parts:

- the message passing subsystem,
- the channel objects and
- the channel manager.

The message passing subsystem provides a GULP message path between channel objects. The channel objects implement the basic GULP state machine and can be subclassed to support different types of devices. The channel manager multiplexes messages and manages the LRU cache of channel objects. The channel manager also keeps a prefetched cache of channel objects that will be used next on new setups. This avoids the costly overhead of page-ins during call setups.

7 Conclusion

The design of *Archos* has been guided by a "keep it simple" methodology. Little differentiation is made between hosts and switch controllers in the network — they both run GULP. The switch code has only to set up circuits when processing a V_CALL message, while the host code has to decide if it should accept a P_INCALL message. The semantics of GULP are not changed in either case, only specialized for each. Commonality has been maximized and differences have been minimized.

The Process/Active Object Per Channel model is expressed as one Channel object managing one channel. Through specialization by subclassing, the SwitchChannel and HostChannel classes share the bulk of the GULP finite state machine code while redefining only a few member functions.

Analysis of Datakit data showed a strong temporal locality in the activity of Datakit channels. Since the Process/Active Object Per Channel model has one Channel object directly corresponding to a channel, the Least Recently Used object replacement algorithm is guaranteed to give a good strategy for maintaining a cache of active objects in the system. To eliminate costly page-ins of active objects during call setup, the system maintains a prefetched cache of objects that will be allocated to new setup requests. In this way, call setups do not have to wait for objects to be paged in before proceeding.

Channel objects are the key components in the *Archos* implementation. Channel defines a common base class for both the switch and host side subclasses, SwitchChannel and HostChannel. They

model the finite state automata defined by GULP and provide a repository for state information of the channels that they govern. Channel objects only react to GULP messages that are sent to them, and only need to execute code when handling a message. ChannelManagers have a pool of processes from which they assign one to the Channel objects when messages arrive for them. Quiescent states are thus multiplexed with a LRU based algorithm. This allows the number of processes in the system to be reduced. The GULP protocol unifies supervision and signalling — dialstring propagation is combined with call setup messages. This streamlines negotiations between two nodes during call setup.

The *Choices* operating system provides an efficient environment for building *Archos*. *Choices* is portable across a wide variety of machines. It currently runs on the Encore Multimax, AT&T 6386E-WGS, IBM PS/2 and Apple Macintosh. Ports to other machines are in progress. *Archos* is thus not hardware specific, but will run on any machine that also runs *Choices*. *Choices* allows *Archos* to have many lightweight processes all executing in the same virtual address space. The message passing system is built with simple shared queues in one address space. *Choices* also allows the dynamic loading of objects into existing code. The controller software can therefore support new devices without requiring that the switch be brought down — the code for a new subclass of Channel is dynamically loaded and linked with the running software. This is an important property in a production system.

Further research in *Archos* will investigate the scheduling of Channel processes among the CPUs in a multiprocessor system to achieve better call setup rates. For example, a pair of communicating Channel processors can be scheduled to execute on different processors, so that no context switching is required between the two. We will also examine network management functions required in a switch controller.

8 Acknowlegements

The authors would like to thank at AT&T: Chuck Kalmenek, Sandy Fraser, Bill Marshall, Gary Murakami and Sean Dorward; at Illinois: the *Choices* group, especially Dave Dykstra and Aamod Sane.

References

[A. 89] A. G. Fraser and W. T. Marshall. Data Transport in a Byte-Stream Network. *IEEE Journal on Selected Areas in Communications*, SAC-7(7):11020–1033, September 1989.

[A+86] G. Abramovic et al. A Conceptual Model for Call Processing. AT&T Bell Laboratories Internal Memorandum, December 1986.

[ATT80] Notes on the Network, Bell System Practices. AT&T Co. Standard Sec. 781-030-100, Issue 2, December 1980.

[CDCH] D. H. Carbaugh, G. G. Drew, H. Chiron, and E. S. Hoover. No. 1 ESS Call Processing. *BSTJ*, 43:2483–2531.

[Che79] G. L. Chesson. Datakit Software Architecture. In *Proceedings of ICC*, June 1979.

[CM90] Roy H. Campbell and Peter W. Madany. Considerations of Persistence and Security in Choices, an Object-Oriented Operating System. In *International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 12.1–12.14, University of Bremen, Federal Republic of Germany, May 1990. The proceedings also to appear in the Springer-Verlag Lecture Notes Series.

[CRJ87] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.

[DC85] N. Q. Duc and E. K. Chew. ISDN Protocol Architecture. *IEEE Communications Magazine*, 23(3):15–22, March 1985.

[Den68] Peter J. Denning. Working Set Model for Program Behaviour. *Communications of the ACM*, May 1968.

[DH82] T. Duncan and W. H. Huen. Sofware Structure of 5ESS — a Distributed Telephone Switching System. *IEEE Transactions on Communications*, pages 1379–1385, June 1982.

[DSS90] S. M. Dorward, R. Sethi, and J. E. Shopiro. Adding New Code to a running C++ program. Technical report, AT&T Bell Laboratories, 1990.

[Ita] Italtel. Proteo UT 10/3 Electronic Digital Exchange. Application Notes.

[Jen85] E. D. Jensen. ArchOS. In *ACM SIGOPS Workshop on Operating Systems in Computer Networks*, January 1985.

[JR91] Ralph Johnson and Vincent Russo. Reusing Object-Oriented Designs: The Choices Virtual Memory Framework, 1991.

[MCK91] Peter Madany, Roy H. Campbell, and Panos Kougiouris. "Experiences Building an Object-Oriented System in C++". In *Proceedings of the Technology of Object-Oriented Languages and Systems Conference*, Paris, France, March 1991.

[McM81] L. E. McMahon. An Experimental Software Organization for a Laboratory Data Switch. *Proceedings of the 1981 IEEE International Conference on Communications*, 2:24.4.1–24.4.4, 1981.

[McM83] L. E. McMahon. TDK Principles of Operation. Technical report, AT& Bell Laboratories, 1983.

[Mey87] Bertrand Meyer. Reusability: The Case for Object-Oriented Design. *IEEE Software*, pages 50–64, March 1987.

[MS90] G. J. Murakami and R. Sethi. Terminal Call Processing with Esterel. Technical Report No. 150, AT&T Bell Laboratories, June 1990.

[Pos80] J. B. Postel. User Datagram Protocol. Technical Report RFC 782, SRI Network Information Center, August 1980.

[PPTT90] Pike, Presotto, Trickey, and Thompson. Plan 9 from Bell Labs. In *Proceedings of the 1990 UKUUG*, 1990.

[PS85] Peterson and Silberschatz. *Operating System Concepts.* Addison-Wesley, second edition, 1985.

[RC89] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[RT75] Dennis M. Ritchie and Kenneth Thompson. The UNIX Time-Sharing System. *AT&T Bell Laboratories Technical Journal*, 57(6):1905, 1975.

[Rus90] V. F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois, Urbana-Champaign, 1990.

[Sch87] M. Schwartz. *Telecommunication Networks: Protocols, Modeling and Analysis.* Addison-Wesley, 1987.

[Tan88] A. S. Tannenbaum. *Computer Networks.* Prentice-Hall Inc., second edition, 1988.