Hardware and Software Architecture of a Packet Telephony Appliance

M. Chan, C. D. Cranor, R. Gopalakrishnan, P. Z. Onufryk,
L. W. Ruedisueli, C. J. Sreenan, E. R. Wagner

AT&T Labs - Research
Florham Park, NJ 07932, USA

Abstract

Networked appliances are easy to use devices that are designed and built for a single function. This paper presents the design and implementation of a networked appliance for packet telephony. We examine the practical issues involved in the design and implementation of the appliance at all layers, including hardware architecture, system services, and the application. At each layer, we identify important challenges and describe the solutions that we implemented. Our packet telephony appliance is built around the Euphony network processor that integrates networking and DSP functions with a CPU. It uses a real-time operating system to provide predictable processing and networking support. In addition to describing the appliance, this paper presents two mechanisms which are of general use in the design of networked appliances. One mechanism is *IObufs*, which provides a unified buffering scheme that allows zero-copy data movement. Another is the Event Exchange (EVX), which provides a flexible mechanism for event distribution, allowing software modules to be composed together in an extensible manner. EVX and *IObufs* are used together to provide highly efficient intra-appliance communication. The combination of EVX and a general-purpose CPU provide a platform that can evolve gracefully to support new protocols, advanced telephony services, and enhanced user interfaces.

1 Introduction

The potential cost savings of using data networks for intra-company telecommunications are well recognized, and many companies are interconnecting their office PBXs using corporate intranets. Given that most office environments are also equipped with high-speed LANs, there is an opportunity for additional cost savings by extending packet telephony to the desktop, thus eliminating the need to maintain two separate office networks. With greater penetration of high-speed residential access and new technologies for in-home networking, similar opportunities exist for using packet telephony in the home.

These developments raise the question of what types of devices people might use to access packet telephony services, motivating the work we present in this paper. It is clear that to capitalize on the new opportunities presented by packet telephony requires devices that can process multimedia data and implement new user interfaces. The use of PCs as telephony devices is hampered by several factors including software/hardware installation and configuration difficulties, low voice quality due to operating system latencies and scheduling idiosyncrasies, and expense in comparison to most consumer appliances. In addition, PCs are relatively large, generate noise and heat, and can be difficult to use. We believe that new special-purpose telephone appliances are the best solution.

In this paper we present the hardware and software architecture of a packet telephony appliance. Our work was guided by four basic principles: low cost, extensibility, ease of use, and reliability. Low *cost* is a crucial factor for consumer devices. Our design addresses this issue through aggressive integration of hardware functions, retaining a basic keypad as the default input device (but supporting remote and attached GUIs), and having low memory requirements by using a small real-time operating system and zero-copy techniques. Our second design principle is *extensibility.* Despite a wealth of experience with packet voice, the field of packet telephony is in its infancy and lacking standards. Thus it is essential that a packet telephony appliance be designed to operate under software control and be able to accommodate new protocols and services as they appear. This is not the case with currently available telephony appliances. We present an event communication mechanism that allows easy integration of new software modules on the appliance. Our third principle is *ease of use*. Telephones must be designed for ordinary people who have no technical background and are unwilling to invest time in setting up and configuring new appliances. By retaining the simple keypad we provide users with a familiar interface. For basic telephony service the appliance needs no user configuration, plug-in components or additional audio equipment. Finally, consumer appliances are expected to operate to a high level of *reliability* despite being used in possibly hostile physical environments and to operate in an "always on" power state. For these reasons we avoided the temptation to use a CPU that requires cooling, used solid state Flash memory rather than a hard drive, and did not include an integrated display. In terms of software, we minimized the phone's reliance on external systems when providing a highly reliable *basic* telephone service.

The goal of our project is to investigate the practical issues involved in designing and implementing a packet telephony appliance. We examine the design space at all layers, including hardware, system services, and the application. At each layer we identify the challenges and propose appropriate solutions. Our contribution is a comprehensive hardware and software architecture that includes a low-cost integrated network processor, a lightweight call signaling implementation, a modular and extensible telephony application design, and an event exchange mechanism for flexible inter-module communication. The Euphony ATM Telephone (EAT) is our custom built packet telephone appliance, shown in Figure 1. It looks much like a conventional telephone and can be used without any training in order to access basic telephony service. It allows many familiar telephony features, such as call waiting, to be implemented locally. It also provides access to networked servers that implement functions such as speech-enabled dialing and call by name.

Figure 1. Euphony ATM Telephone

In the next section we describe the system environment in which EAT operates and elements of our packet telephony infrastructure. Section 3 describes hardware architecture. Section 4 discusses software system services while Section 5 presents the telephony application. Sections 6 & 7 discuss related and future work. Finally, Section 8 offers conclusions.

2 Packet Telephony Environment

There are many ongoing activities in the standards and research communities that relate to packet telephony. For example, the ITU has defined the H.323 [9] family of protocols for multimedia communication, and the IETF is standardizing the Session Initiation Protocol (SIP) [8] as a mechanism for setting up calls. At AT&T Labs, the Telephony Over Packet Networks (TOPS) [2] architecture addresses several aspects of packet telephony. Many key issues are still topics for research, including quality of service, security, privacy, authentication, and billing. Fortunately, the functional components of a packet telephone are largely independent of the underlying networking technology, signaling, and directory services employed. The packet telephony appliance described in this paper is based on the TOPS architecture, but is designed to easily evolve to support new standards. To aid in understanding the functionality required of our packet telephone, we provide a brief overview of the TOPS architecture.

A guiding principle of TOPS is to make it more convenient to reach a user. A packet telephony service requires a directory service function that can translate between telephone numbers and network layer addresses, such as IP or ATM addresses. This simple functionality is enhanced in TOPS to allow callers to reach a person using a Distinguishing Name (DN) rather than an address by storing information in the directory service about DNs and the set of devices where users can be reached. A DN is a unique identifier for the person or entity to be called and may be an X.500 distinguishing name, an e-mail address, or a traditional telephone number. Callers obtain terminal addresses of users by issuing a name resolution query to the directory service. As users move between terminals, they can securely modify their record entries to reflect the locations where they can be reached. Thus, user mobility is supported as a fundamental capability of TOPS.

The directory service provides a mechanism for individual users to control how a name resolution query is handled, for example based on time of day or caller identity. Thus, users can customize access to their information. Compared to existing telephony features such as "personal phone numbers" and "intelligent 800-number" services, this approach is more flexible while also giving more control to the user. TOPS terminals range from computers running telephony applications to low-cost packet telephony appliances. Terminal capabilities are stored in the directory and returned to the caller as part of name resolution so that appropriate communication resources can be setup.

Functions provided by the traditional telephone network, such as routing, connectivity, and resource management, are already supported by packet networks. This, coupled with intelligent end-systems reduces the need for intermediate network entities to be involved in packet telephone calls. Packet telephones in the TOPS architecture communicate directly with each other using Application-Layer Signaling (ALS). ALS is used to setup a call, negotiate capabilities, establish and manage media channels, and terminate calls. TOPS defines an efficient encapsulation format for audio data but does not preclude the use of other formats.

We have constructed a packet telephony testbed to gain experience with the TOPS architecture. To ensure good quality voice transport, we adopted ATM in our testbed as the underlying networking technology. Voice is carried over switched virtual circuits which are established at call time. Other data such as call signaling and directory interactions do not have strict delay requirements and for simplicity are implemented using IP over ATM. Our environment includes a gateway providing a bridge to a PBX. We have several other terminal types in addition to EAT. These are: a PC-equipped with a sound card running a telephony application, a PC acting as a proxy for a set of analog telephones, and a voice-enabled wireless PDA [6].

3 Hardware Architecture

The goal of prototyping EAT was to demonstrate that a packet telephony appliance can be constructed that implements the advanced services promised by packet telephony and does so at a cost in line with consumer devices. Reliability and ease of use were also important goals of the prototype. The central component of EAT is Euphony [15,16], a network processor which integrates computing, media processing, and networking into a single low-cost VLSI device. Euphony was jointly developed by AT&T Labs - Research and LSI Logic and implemented using LSI Logic's LCB500K 0.5 micron drawn cell-based ASIC process. Euphony integrates many of the functions required to build network appliances on-chip, thereby reducing cost, power, and size.

Euphony is based on a MIPS RISC processor core that was augmented with a single cycle pipelined multiplier and signal processing instructions. Most of the system logic required to build a complete system is contained within Euphony, including the logic required to interface to standard SRAMs, DRAMs, VRAMs, and many peripheral devices. Power-on reset generation and a five channel DMA controller are provided on-chip. In addition, Euphony provides general purpose I/O pins which may be configured as bit I/O ports or as interrupts.

Euphony contains two primary I/O interfaces, a serial audio port compatible with many popular A/Ds, D/As, and telephone codecs and an ATM interface with architectural support for AAL5 segmentation and reassembly (SAR) processing. Unlike traditional SARs which are either completely implemented in hardware or software, Euphony's ATM interface provides hardware support for time critical functions, such as CRC-32 calculation, and leaves non-time critical functions to software. The advantage of this approach is that it provides good performance and only requires a small amount of die area.

A block diagram of EAT is shown in Figure 2. EAT contains 4 MB of SRAM and 2 MB of Flash memory[1]. Although EAT is targeted as a cost sensitive consumer device, we chose to prototype the system using SRAM since it allowed us to experiment with different memory speeds by changing the number of wait-states used in SRAM accesses. A commercial version would use DRAM connected to Euphony's DRAM controller.

Euphony's signal processing capabilities coupled with its integrated serial port provides a good platform for packet telephony audio processing. EAT has three audio input/output interfaces: a telephone handset, a case mounted microphone and speaker, and an external microphone/line input and line output. The case speaker is used to generate telephone ringing. This allows EAT to not only generate traditional telephone ringing tones, but also allows the implementation of

1. Our packet telephony application together with all the system code occupies about 2MB of SRAM and 1MB of Flash memory.

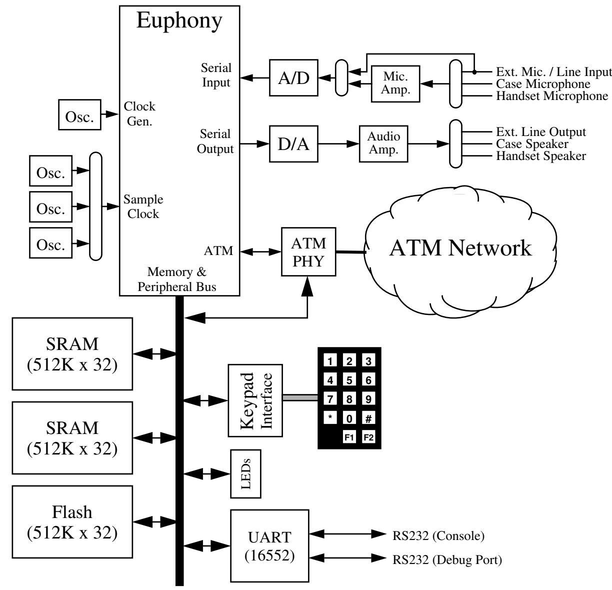

Figure 2. EAT Block Diagram

services such as voice announcement. The case microphone and speaker together may be used to implement a speaker phone. The external microphone/line input and line output support two audio channels (i.e., stereo operation) and allow EAT to connect to external speakers and microphones.

The audio input and output interfaces are implemented by connecting Euphony's serial port to high quality 18-bit linear A/D and D/A converters. Audio data format conversion, such as μ-law, is performed in software. Two of Euphony's DMA channels are used to move data between memory and the A/D and D/A converters. The sample rate for the A/D and D/A may be independently selected to be 44.1 KHz, 32 KHz, or 8 KHz.

EAT is implemented using two circuit boards mounted within a custom built plastic case. The system logic board, shown in Figure 3, mounts in the base of the case and contains all of the digital and analog components. A second I/O interface board containing a keypad, four status LEDs, and a micro-switch for the off-hook detector mounts at the top of the unit. The system logic board and I/O interface board are connected with a ribbon cable.

EAT connects to standard 25 Mbps ATM networks using Euphony's ATM interface. In addition to an ATM connection, EAT contains two RS232 serial ports, a console and a debug port. Only the console is available for external connections. A PC-style DB9 connector on the back of the case can be connected to external devices, such as a Palm Pilot or an LCD touch display.

4 EAT System Software Services

In this section we examine the system-level software services provided by EAT. These services consist of a real-time single

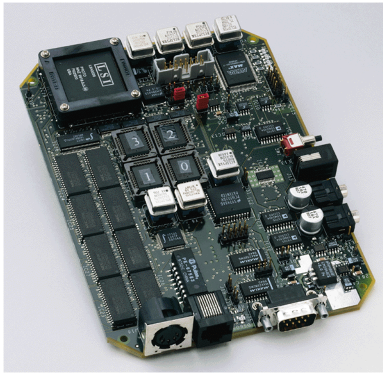

Figure 3. EAT System Logic Board

address space operating system kernel, a zero-copy I/O buffering mechanism used for communication within EAT, an event-based mechanism for inter-module communication, and an IP/ATM networking stack. Figure 4 shows EAT's system software environment.

4.1 Kernel

EAT loads and executes the VxWorks kernel on power-up. Once booted, the VxWorks kernel can load applications from a network server or from an onboard Flash file system. There are several reasons why we chose VxWorks for EAT rather than a standard operating system such as Linux. First, the VxWorks kernel has mature support for multithreaded applications in an embedded systems environment while systems such as Linux are optimized for a timesharing environment. Second, VxWorks supports the real-time scheduling features that our telephony applications require. Third, VxWorks has a smaller memory footprint than traditional systems because general purpose kernel features such as disk-based filesystems, virtual memory, multiple address spaces, multiuser support, and user accounting are not relevant to the EAT environment. In addition, like standard operating systems, VxWorks provides a rich environment that supports dynamic loading of application code and a socket-based TCP/IP networking stack, but unlike standard operating systems which are self-hosting, VxWorks provides an excellent set of tools for embedded systems development.

VxWorks has a single address space in which the kernel and application threads execute. The thread scheduler supports both timesharing and preemptive scheduling based on static real-time priorities. VxWorks supports interprocess communication by providing pipes, message queues, and semaphores. Threads blocked on a semaphore can queue in priority order. To minimize priority inversion for real-time threads, VxWorks implements priority inheritance for semaphore operations.

4.2 *IObufs*: A Uniform Buffering Mechanism for Zero Copy Operation

To reduce memory usage and to avoid the additional latency of data copying, we aggressively use copy reduction tech-

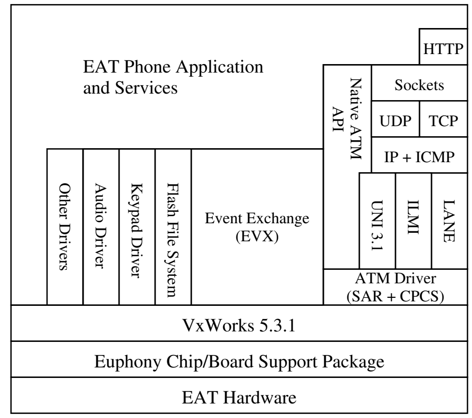

Figure 4. EAT Software Architecture

niques by taking advantage of EAT's single address space architecture. Our approach to achieving zero-copy is the use of *IObufs* for storing and passing data. *IObufs*, whose structure is shown in Figure 5, are similar to "mbufs" used in 4BSD Unix systems [11]. Like mbufs, *IObufs* allow buffer manipulation operations, such as linking to form larger packets, without data copying. In addition, application specific information can be stored within *IObufs*. Our ATM driver uses this feature to store DMA state and partial CRC checksum information while transferring data between *IObufs* and the network. The novel aspect of our system is the use of *IObufs* as a uniform buffering mechanism across all modules in EAT including the application and I/O subsystems. This reduces data movement costs and enables cross subsystem optimizations. In addition to this, we use *IObufs* with the event exchange IPC mechanism described in the next section to obtain benefits such as one to many communication and flow control. Thus the combination of these two mechanisms achieves integration of event distribution with data movement.

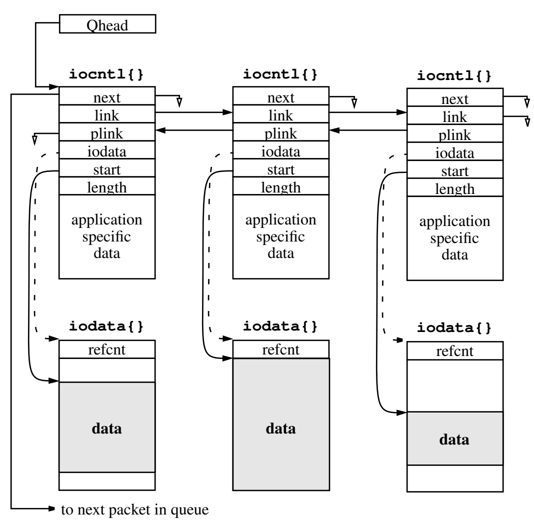

Figure 5. IObuf Chain

4.3 EVX: An Inter-Module Communications Mechanism

EAT applications consist of a number of interconnected software modules. These modules come in two classes: those which interface with hardware devices and those which implement software services. The initial EAT software environment did not define a mechanism for inter-module communication. As a result, modules shared information and synchronized themselves in an ad-hoc manner using VxWorks interprocess communication mechanisms. This led to three unfortunate limitations. First, modules were tightly coupled making it difficult to reconfigure interconnections without editing affected modules and recompiling. Determining module interconnection was also a challenge, since predefined module calls were distributed throughout the application. Second, our initial applications had a restricted form of event delivery; an event producer could only deliver an event to a single consumer. This is a limitation because we found that multiple modules needed to receive the same types of events, for example a keypad event needed to be sent to both a tone generator module and a digit collection module. Third, the lack of structured inter-module communication coupled with EAT's unprotected single address space blurred the boundary between modules by allowing them to communicate by modifying each others global memory state rather than using a well defined API. This later caused us problems when reconfiguring and debugging EAT applications.

To address these problems, and to provide a more flexible software environment for EAT applications we introduced the Event Exchange (EVX) inter-module communication mechanism. The Event Exchange allows components of the phone to share information in a flexible and efficient manner. EVX delivers events posted by a module on its "sending port" to one or more interested modules on their EVX "receiving ports." Features of EVX include:

- Events are named independently of module function names or addresses allowing easy reconfiguration of EVX-based applications. A module can be replaced without modifying other modules.
- Events can be delivered to one or more receivers without the event producing module having to explicitly identify the event's receivers.
- Event data is delivered through a zero-copy reference count-based mechanism that reduces the cost of delivering an event to multiple modules. This mechanism can be used with *IObufs* to provide a way to exchange bulk data at low cost.
- Events are queued at the sending port to provide flow control. This prevents event producers, such as the tone generator, from consuming all of EAT's resources.

EVX allows EAT-based applications to be separated into two parts: a set of independent modules and a small compositional application that ties modules together. To use EVX, a developer determines which modules are needed for the application and how they interconnect. In the current version of EVX, interconnections are determined when the compositional application is written, but future versions will allow for runtime configuration. When the compositional application starts, it initializes EVX's global state and creates sending and receiving ports for each module. Each port has a character string name and an associated port data structure. EVX manages this association so that function names do not have to be hardwired into modules. Each module commences by initializing its ports before use, using the queue size of its sending port to provide flow control and to prevent the module from exhausting memory resources. At that point, event producing modules can post events to their sending ports. Posted events are delivered to receiving modules by EVX. Receivers process an event and then issue an acknowledgment. EVX allows threads to block waiting for an event and/or arriving network data.

EVX Function	Description
evx_init	Initializes EVX's global state.
evx_connect	Establishes a connection between a sending port and a receiving port. Ports are referenced by their string names. Data structures are allocated when the port is referenced in a evx_connect call. This includes setting the port's queue length.
evx_initsp	Initializes a sending port.
evx_initdp	Initializes a receiving port.
evx_post	Post an event to a sending port. This blocks if the port's event queue is full and the non-blocking flag is not set. Optionally, evx_post will wait until the event is delivered, providing synchronous semantics.
evx_receive	Receive an event from a sending port. If there is no pending event and the non-blocking flag is set, then evx_receive returns null. When a receiver finishes processing an event it must call evx_ack to acknowledge it.
evx_ack	Release a receiver's reference to a received event, allowing the sender to recycle the event for future use. Optionally the sender may request a callback when the event is freed.
evx_swalloc	Allocate a new semaphore-wait object. These objects allow a thread to block waiting for events on the associated port(s). An object can be associated with a pipe, allowing a thread to use `select` to simultaneously wait for an event and I/O.
evx_swadd	Add a receive port to a semaphore-wait object. A receive port is associated with only one object at a time.
evx_swremove	Remove a receive port from a semaphore-wait object.
evx_swwait	Wait for an event to arrive on one of the receive ports added to the specified semaphore wait object.
evx_swget	Get a list of receive ports that have pending events from a semaphore-wait object.
evx_swpipefd	Return the pipe file descriptor associated with the specified semaphore-wait object so that a client may `select` on it. EVX writes a byte to the pipe when a receive port with no queued events receives one. EVX removes this byte from the pipe when an event is received.

Table 1. EVX API

An important property of the EVX event distribution mechanism is that events are processed at the priority of the receiving thread. This is motivated by the need to support predictable processing for EAT application modules that deal with media streams such as audio. Decoupling the priority of the sender from the priority at which the event is processed

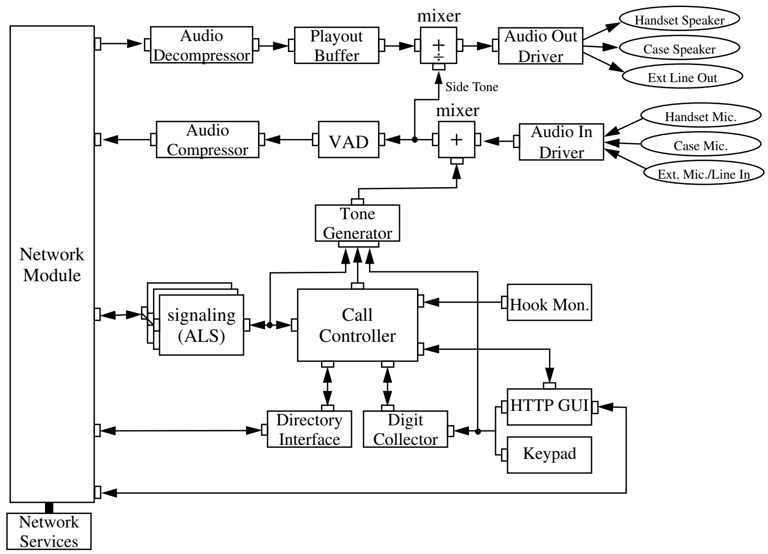

Figure 6. Modules in EAT Packet Telephony Application

naturally lends itself to a multicast IPC model since the same event can be processed at different priorities depending on the receiving thread.

Table 1 shows the EVX API. The main functions used by EVX modules are `evx_post` to post an event, `evx_receive` to receive an event, `evx_ack` to acknowledge a received event, `evx_swwait` to wait for an event to arrive, and `evx_swget` to get a list of receive ports with pending events. Note that the use of semaphore-wait objects allows a decoupling between the specification of events of interest from waiting for events to arrive, thus addressing the known problem with `select` in scaling to a large number of descriptors [3].

Adapting the EAT application to use EVX has yielded reduced complexity and easier reconfiguration and extensibility. For example, it allows signaling-specific functions to be encapsulated into a single module. To use a different signaling protocol just requires adding a new signaling module and adjusting module interconnections. In the future, combining dynamic EVX reconfiguration and dynamically loaded code will provide even more flexibility to EAT application developers. Section 6 describes how EVX is used in the EAT packet telephony application.

4.4 Networking

EAT supports both IP over ATM and native ATM networking. EAT takes advantage of the flexibility and variety of services offered by IP to implement control functions and uses ATM to provide network QoS for voice traffic. As shown in Figure 4, EAT networking is comprised of an ATM driver that implements segmentation and reassembly, ATM layers, and a TCP/IP stack.

EAT implements segmentation and reassembly with a combination of hardware and software. The hardware performs most of the work: cell header checking, DMA with CRC calculation, and physical cell transmission and reception. The software initiates DMA transfers for cell transmission and reception. Each AAL5 packet is comprised of one or more *IObufs,* allowing software to simply hand off a pointer to an *IObuf* to initiate a DMA transfer. The software also does cell pacing.

The ILMI, UNI, and LANE ATM layers are part of the Harris and Jeffries (H&J) Soft ATM package. ILMI registers EAT's ATM address with the switch while UNI provides signaling for connection setup and release. LANE allows an Ethernet physical layer to be emulated on an ATM network. When used with an Ethernet bridge, LANE allows EAT to communicate over both ATM and Ethernet.

The VxWorks TCP/IP stack on EAT provides a complete set of network facilities. This allows remote file access, remote procedure calls, and access to EAT using telnet. The network facilities also include WindRivers embedded HTTP Server. We use the HTTP server to implement a GUI for advanced services. Using a web browser, a user could place a call, check messages, perform directory lookups, or change EAT configurations.

5 EAT Packet Telephony Application

The EAT packet telephone application is a program that runs on top of EAT's hardware and software services described in the previous two sections. Figure 6 shows major components of the application and their interconnections using the EVX event communication mechanism.

5.1 Mixer

A mixer module accepts *IObufs* containing audio samples posted as EVX events on its two receive ports and produces an EVX event with an *IObuf* containing the summed digital samples on its send port. The mixer may scale audio samples from its ports prior to summing them. Scaling factors can be set on a per port basis by the call controller.

The EAT telephone application contains two mixers. The mixer in the audio output path mixes audio samples received from a microphone with audio samples produced by a tone generator. This allows tones resulting from key presses to be heard by the remote endpoint. The mixer in the audio input path sums audio samples received from the network with a scaled version of the locally generated audio samples, and feeds the result into the audio output device. The scaled version of the locally generated audio is called the *side-tone*. Side-tone is auditory feedback that enables the speaker to hear her own voice. Absence of side-tone makes it difficult to determine how loudly to speak, and gives the phone a dead feeling.

5.2 Tone Generator

In the current telephone network, tones are used to provide various forms of auditory feedback to the user and as a form of in-band signaling. For example the *busy tone* is used to notify the caller that the far end is busy. A combination of tones are used to represent keypad symbols. These tones are often converted back to keypad symbols by a tone detector at the far end in systems such as voicemail. Although a packet network provides separate data and control channels, touch tones are still useful when interworking with legacy systems.

Rather than increasing the cost of EAT by using a special purpose hardware tone generator, EAT implements tone generation in software. We wrote a fixed-point integer-based tone generation module which has low overhead. The EAT tone generator produces tones in response to EVX events posted on its receive port. Tone samples are placed into *IObufs* and are posted as events on its send port. The EVX flow control mechanism allows the rate at which the tone generator produces *IObufs* to adapt to the rate at which the D/A drains audio samples.

When the phone application is started, the tone generator module creates a 512-entry table consisting of equally spaced samples of a sine wave which are scaled for full output. To generate a tone of a particular frequency f, the tone generator treats the table as a circular array and samples it every $\left\lceil 512\left(\frac{f}{s}\right)\right\rceil$ entries, where s is the sampling frequency. Clearly, this approach produces tones which are not exact. We chose a table with 512 entries since it requires only a small amount of memory (2048 bytes) and allows tones to be generated with small enough error to interoperate with legacy telephone systems.

Multiple single frequency tones must be summed to create most standard telephone tones. For example, the tone produced by pressing key 5 on a conventional telephone is equal to the sum of a 770 Hz tone and a 1,336 Hz tone. The table is also used by the tone generator to produce more complex tones such as busy and ringing.

5.3 Compression

Packet telephony audio is typically compressed before being sent out onto the network. There are several well known standards for audio compression. In EAT we currently use G.711 (μ-law), since it is simple to implement and has negligible processing overhead — encoding and decoding only require a table lookup and a few shift and logical operations.

Audio compression is implemented using two modules, an audio compressor and an audio decompressor. The audio compressor module accepts *IObufs* containing 18-bit PCM samples on its receive port. It converts the PCM samples to 8-bit μ-law and stores them in *IObufs* that are posted as events on its sending port. The audio decompressor module performs the reverse operation.

Although EAT currently implements G.711, the audio compressor and decompressor modules could be replaced with modules that implement other standards, such as G.728. We are planning to modify the phone application to allow the call controller to select the one of several coding standards negotiated during call setup.

5.4 Silence detection

Besides compression, another way to reduce network bandwidth is silence suppression — when a speaker is silent, voice packets are not transmitted. Detecting silent periods is done with a Voice Activity Detector (VAD). The challenge in implementing a VAD is being able to determine what parts of an audio signal are voice and what is background noise which can be treated as silence. A VAD operates by computing the power of each discrete sample and comparing it with a decision threshold. It adjusts the threshold in response to changes in noise levels. For EAT we adopted the VAD described in [10] primarily because it is less complex than existing approaches and requires less numeric precision for computing and updating operating parameters.

5.5 Playout Buffer

The playout buffer module receives decompressed audio samples from the network and performs delay jitter removal and loss concealment. Audio packets can experience variable transit delays due to queuing and congestion. The variance in delay is called the jitter. The playout buffer module estimates jitter and delays audio samples appropriately. This introduces additional delay but avoids gaps during audio playback caused by audio packets arriving late. Packet networks can also occasionally lose packets. Although it is impossible to reconstruct audio samples from a missing packet, it is possible to produce transitions that are less distracting than pure silence. Loss concealment mechanisms are available to do this.

5.6 Putting The Audio Path Together

The audio input driver delivers audio samples from the D/A in *IObufs*. The *IObufs* are passed as EVX events to the mixer in the audio output path. The mixer sums these audio samples with samples from the tone generator. The output of the mixer is then sent to the audio input path as side-tone and to the VAD module. VAD passes non-silent audio samples to the audio compression module for coding. Finally, once a fixed number of coded samples are available, they are passed to the network interface module where they are encapsulated in a packet and transmitted on the network. We currently employ a simple encapsulation similar to that described in [5] but in the future other encapsulations, such as RTP [21], may be used.

The network interface module receives audio packets from the network, decapsulates them, determines if any packets were lost, and passes the audio data in *IObufs* as EVX events to the audio decompressor module. There the audio samples are uncompressed and passed to the playout buffer module. The playout buffer module determines the playout time for the audio samples. If packets were lost, the playout buffer module performs loss concealment. When it is time for data in the playout buffer to be processed, an *IObuf* is sent to the mixer in the audio input path. This mixer combines audio received from the network with a scaled version of the side-tone and passes it to the audio output driver. The audio output driver plays the audio on the selected output device (i.e., handset, case speaker, or external speaker).

5.7 Call Controller

The call controller sets up and manages telephone conversations on the packet telephone. When a user initiates a new call, an EVX event is sent by a user interface component (e.g. the hook monitor) to the call controller. In response to this, the call controller prompts the user for the distinguished name (DN) of the party to be called. Since the DN can be input using any available user interface such as a keypad or a remote web browser, we use EVX events to prompt the user and to pass the DN to the call controller. Thus, different user interfaces can be accommodated.

Once the call controller has obtained the DN, it initiates a directory lookup to map the DN to a set of call appearances. The call controller passes information from a call appearance as an EVX event to a signaling module which uses the event information to establish a call. Unlike most traditional phones, a packet phone may have multiple simultaneous calls in progress. It is the function of the call controller to pass user interface events it receives to the currently active call context (i.e., the one in focus).

5.8 Signaling

At present, our signaling module implements the TOPS application layer signaling (ALS) protocol [2]. For each call, an ALS module maintains a call state machine and makes the appropriate transitions according to network messages and user actions that it receives on EVX ports. It uses EVX events to post call states that are of relevance to other components. For example when ALS gets a busy indication from a called party it posts this state as an event on its send port. This causes the tone generator to generate a busy tone if that call is in focus. Using a common EVX event format for posting signaling events allows ALS to be replaced by different signaling protocols without having to modify other modules in the application. It also allows a single packet telephone application to support multiple signaling protocols on a per call basis.

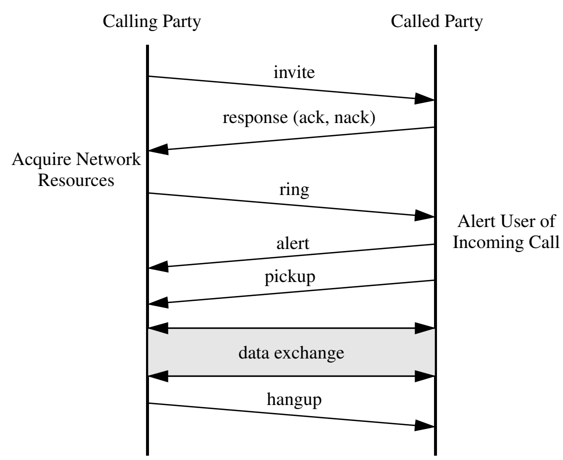

Figure 7. ALS Protocol Exchange

ALS was designed to allow telephony applications and servers to interact for the purposes of call establishment, management, and termination. A principle in its design was that the common case of point-to-point voice telephony must be supported simply and efficiently, while more complicated scenarios, such as multimedia and conference calls, can be handled through protocol enhancements. Salient features of the protocol are:

- A lightweight protocol with a small set of messages,
- Single protocol for all interactions (i.e., between applications, PSTN gateways, and conferencing servers),
- Allows negotiation of call parameters such as number of media streams and terminal capabilities,
- Allows interaction with network protocols to ensure that sufficient resources are available for a call,
- Transport-layer independent.

ALS uses a two-phase message exchange. The first exchange involves inviting the remote terminal to enter into a call and includes call parameters and terminal capabilities. If the remote terminal does not support the requested capabilities, or the remote terminal is busy, a negative acknowledgment is returned. A positive acknowledgment causes media streams to be set-up and network resources to be allocated. If resources are available, a second message exchange is used to alert the called party of an incoming call. Figure 7 shows these message exchanges. Figure 8 shows a single call protocol state machine at the calling party. For simplicity, some states, unexpected conditions, and time-outs are not shown.

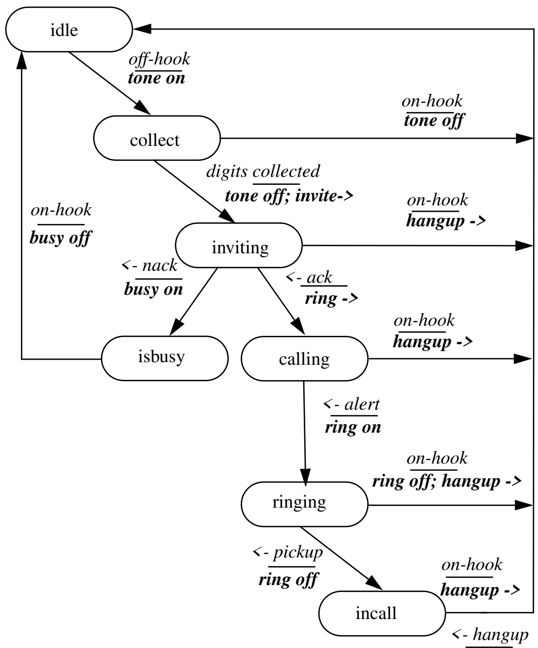

Figure 8. Calling Party State Machine

6 Related Work

Numerous techniques exist to reduce copying in the network subsystem. The closest scheme to *IObufs* is I/O-Lite [17]. Both systems use a common data representation across subsystems but I/O-Lite requires that buffers be immutable, since it uses this property to enforce protection when data is shared between domains. *IObufs* avoids this problem since all threads are in the same domain. *IObufs* can be located in any memory region. In contrast, I/O-Lite uses the Fbuf [4] mechanism for data movement and its buffers are restricted to be in a specific region. This can result in data copying.

EAT's event exchange (EVX) provides a software bus-style mechanism that integrates event delivery with *IObuf* data communication. EVX is based on the publisher/subscriber model [14,18] and naturally supports one-to-many communication. By defining module communication in terms of send and receive ports, EVX obtains the flexibility of distributed object systems, and can support similar approaches for decoupled event delivery [7]. EVX provides flow control similar to Unix SVR4 STREAMS [19,23]. Scout [12,13] is an example of a operating system that targets network devices. It uses the notion of a *path* that connects modules and allows resource accounting and quality of service, but does not address integrated event/data delivery.

EVX's integration of networking and event notification through the use of a pipe is similar to the mechanism used in [20] to hide the difference between semaphores and socket-based I/O. EVX's separation of event interest selection and waiting for event notification was based on observations in [3] on the expense of using `select` to wait for I/O.

Our work draws on work conducted during the 1980s on projects such as Etherphone [22] and ISLAND [1], which explored many of the basic issues related to packet voice transmission using voice terminals in an office LAN environment. Since we began this project, several companies have released IP telephone products targeted at the LAN-based PBX market. For example, Selsius Systems Inc. provide an Ethernet-based phone and Symbol Technologies Inc. offer a wireless LAN phone. These phones are designed to use a predetermined protocol (H.323) and fixed coders, and rely for the most part on a LAN-based PBX server to implement telephony services. In comparison, EAT was designed specifically to be extensible and can accommodate new protocols and services easily.

7 Future Work

EAT is now an integral part of the TOPS packet telephony environment at AT&T Labs. In addition to augmenting the browser-based control interface, we plan to use EAT to explore several other issues related to telephony appliances. An interesting feature of EAT is its ability to implement telephony services such as call waiting at the telephone device itself, rather than in network switches, as is the case for traditional telephony. This allows new services to be more easily tested, implemented, and deployed. In designing EAT we provided mechanisms that enable extensibility, and we plan to explore how new services can be automatically discovered and dynamically installed in a phone. For example, some form of service scripting language may be appropriate. EAT programmability may also be used to dynamically load voice coders based on call capability negotiation and to provide personalized behavior based on the preferences of a registered user. How to provide these features is a subject of future work.

Telephony appliances are required to interact with various network servers, for example EAT currently uses the TOPS directory service and a gateway. We are interested in exploring more generally how EAT might use other services to support conferencing, speech processing, and user/terminal/call mobility. Finally, we are interested in using EAT to enable new services, including those involving access to high-quality music and the use of location information to influence call handling.

8 Conclusion

In this paper we presented the design and implementation of EAT, a packet telephony appliance. Our work makes three contributions: the detailed system architecture of a packet telephony appliance, a unified buffering mechanism for space and time efficiency, and a related event exchange mechanisms that enables extensibility. Our paper describes the overall packet telephony system architecture. The EAT packet telephony appliance that we built is based on the novel Euphony network processor that integrates networking and DSP functions to provide a low cost and efficient solution for building networked appliances. EAT runs a real-time operating system that allows predictable operation with support for guaranteed processing of voice data. The system services provide for buffering, interprocess communication, and networking on which the phone application is built. Finally, we presented the architecture of the phone application itself and the interconnection between its components.

In addition to describing the hardware and software architecture of EAT, our paper introduced two general mechanisms which grew out of our need to conserve memory and a highly efficient but extensible software system. The first of these is *IObufs*, which are used by all software modules that require data to be exchanged. The second is the Event Exchange. The combination of these two mechanisms provides an efficient scheme for integrated event/data delivery and allows EAT to easily evolve to accommodate new protocols and services.

References

[1] S. Ades, R. Want, R. Calnan. "Protocols for Real Time Voice Communication on a Packet Local Network",Proceedings of IEEE ICC, pages 525-530, June 1986.

[2] N. Anerousis, R. Gopalakrishnan, C.R. Kalmanek, A.E. Kaplan, W.T. Marshall, P.P. Mishra, P.Z. Onufryk, K.K. Ramakrishnan, C.J. Sreenan. "TOPS: An Architecture for Telephony Over Packet Networks," IEEE Journal on Selected Areas in Communications (JSAC), 17(1):91-108, January 1999.

[3] G. Banga, and J. Mogul, "Scalable Kernel Performance for Internet Servers Under Realistic Loads," Proceedings of USENIX, pages 1-12, 1998.

[4] P. Druschel, and L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, pages 189-202, 1993.

[5] A. G. Fraser, P. Z. Onufryk, K. K. Ramakrishnan, "Encapsulation of Real-Time Data Including RTP Streams over ATM", Included in H.323 Media Transport Over ATM Living List of Issues in the ATM Forum, February 1998

[6] S. Goel, P. Mishra, H. Saran, C. J. Sreenan. "Design and Evaluation of a Platform for Mobile Packet Telephony," Proceedings of IEEE International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), pages 71-81, July 1998.

[7] R. Gruber, B. Krishnamurthy, and E. Panagos, "High-Level Constructs in the READY Event Notification System," Proceedings of the Eighth ACM SIGOPS European Workshop, 1998.

[8] M. Handley, H. Schulzrinne, E. Schooler. J. Rosenberg. "SIP: Session Initiation protocol," IETF draft <draft-ietf-mmusic-sip-12>, work in progress, January 1999.

[9] ITU-T. "Recommendation H.323: Visual Telephone Systems and Equipment for Local Area Networks Which Provide a Non-guaranteed Quality of Service," 1996.

[10] D. Malah. "A Novel Approach for VAD Adaptation in Nonstationary Noise Environments," AT&T Labs Technical Memorandum, Doc. No. HA6153000-981015-09TM, October 1998.

[11] M. McKusick, K. Bostic, M. Karels, and J. Quarterman, "The Design and Implementation of the 4.4BSD Operating System," Addison Wesley, 1996.

[12] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, and T. Proebsting, "Scout: A Communications-Oriented Operating System," Proceedings of HotOS, pages 58-61, 1995.

[13] D. Mosberger, and L. Peterson, "Making Paths Explicit in the Scout Operating System," OSDI, pages 153-167, 1996.

[14] B.Oki, M. Pfluegl, A. Siegel, D. Skeen, "The Information Bus–An Architecture for Extensible Distributed Systems," ACM Symposium on Operating Systems Principles, 1993, pages 58-68.

[15] P. Z. Onufryk, "Euphony: A Signal Processor for ATM," EE Times, pages 54-80, January 20, 1997.

[16] P. Z. Onufryk, "Euphony: An Embedded RISC Processor for Low Cost ATM Networking and Signal Processing," Design SuperCon97: Digital Communications Design Conference, pages C112-1 to C112-16, 1997.

[17] V. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A Unified I/O Buffering and Caching System," Proceedings of the Third Symposium on Operating Systems Design and Implementation, pages 15-28, February 1999.

[18] R. Rajkumar, M. Gagliardi, L. Sha, "The Real-Time Publisher/Subscriber Inter Process Communication Model for Distributed Real-Time Systems:Design and Implementation," IEEE Real-Time Technology and Applications Symposium, June 1995.

[19] D. Ritchie, "A Stream Input-Output System," AT&T Bell Laboratories Technical Journal, 63(8):1897-1910, 1984.

[20] D. Schmidt, "The ADAPTIVE Communications Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications," Proceedings of the 12th Sun Users Group, June 1994.

[21] H. Schulzrinne. "RTP: Real-time Transport Protocol," RFC 1889, January 1996.

[22] D. Swinehart, L. Stewart, S. Ornstein. "Adding Voice to an Office Computer Network", Proceedings of IEEE GlobeCom, November 1983.

[23] USL, "Unix System V Release 4.2 STREAMS Modules and Drivers," Prentice-Hall, 1992.