GNU Bayonne: telephony services for freely licensed operating systems

David Sugar <*sugar@gnu.org*>
http://www.gnu.org/software/bayonne

Abstract

In understanding the needs of building current and next generation telephony services we have to consider that in the past such services were often built from proprietary realtime operating systems. Changes in the last 20 years in the power and capabilities of commercial off the shelf commercial platforms and free operating systems have largely replaced the need for specialized proprietary realtime systems, but not the realtime requirements that still exist today. This paper is meant to provide an overview and introduction of the challenges and needs in providing realtime Linux services to support current and next generation telephone networks, but is not meant to be a rigorous analysis on the topic.

1 Introduction

Even without considering all the various reasons of why we must have Free Software as part of the telecommunications infrastructure, it is important to consider what the goals and platform needs are for a telephony platform. Historically, telephony services platforms had often been the domain of hard real-time operating systems. While it is true recent advances in computer telephony hardware has made it possible to offload much of this requirement to hardware making it practical for even low performance systems running efficient kernels to provide such services for many concurrent users, this has not eliminated issues related to realtime services.

While hardware has improved much, new technologies, such as wide deployment of packetized voice at the end user level, have also, in fact created a whole new set of realtime constraints, and these need to be addressed by modern freely licensed operating systems wishing to be used for telephony applications. Finally, with the ever increasing power of cpu's, there has been a move to simplify commodity computer telephony hardware by offloading dsp processing back to the host cpu.

All of these changes effect the role that realtime Linux kernels can have to play in current and next generation telephone networks. These roles, as we will explore, are not limited potentially just to servers. Realtime linux systems can also have a role to play in modern computer telephony hardware and of course in embedded communication devices.

2 Recording Audio

People communicating to some extend are realtime systems from the perspective of the telephone network. The telephone network cannot impose flow control on human conversation, for example, should it need to for pausing. While full duplex conversations are often passive connections in telephone switching systems, there are places where realtime constrains become more visible, and this is particularly true in automated systems where voice is being recorded, such as when storing voice messages for voice mail systems.

Human speech is almost universally encoded as either a-law or u-law audio by the public telephone networks. The quality of copper circuits vary, but the old telephone network was never designed for carrying high bandwidth or high fidelity audio over copper. In fact, 8 bit pcm encoded audio, at 8khz sample range, is about the limit one can expect, and some places do not even achieve this.

When recording audio, then, this means that for every second of audio, 8K of data must be stored somewhere. In many older systems, even this modest requirement can be a challenge, especially if it needs to be done for several hundred different concurrent sessions. Disk bandwidth of older IDE drives (and older SCSI systems) was typically in the range of 100 to 200Kbytes of data per second, and before that, there was MFM systems with the then standard drive interface that did even poorer.

When one talks about recording concurrent voice, one is also talking about being able to do so from multiple sessions, and most ancient drives, in the days before cache, would have both rotational and seek latencies that would further delay the ability to write voice data timely from current sessions. Outside of SCSI, most ancient drives could not do multiple I/O drive requests.

In this environment, while a comfortably 100-200k of potential bandwidth existed for writing of voice files, in fact the actual disk I/O performance might restrict actual bandwidth achieved by a magnitude. In that most early hardware was ram miserly, and most early systems had little ram available for pre-buffering of audio before recording to disk, this was often a great technical challenge. Considering that humans cannot be flow controlled, and 8k per channel per second needed to be recorded regardless of all these limitations, this is clearly a task that could be scheduled and defined by a deterministic realtime system, and many clever things were done in systems in the past to get around these storage performance issues.

Certainly there were things that could be done to improve this scenario starting with voice compression. Most early hardware was fairly limited in what it could do, but basic linear codec's such as 3-4-5 bit ADPCM can and were used for sampling speech to be recorded, and often at a reduced sample rate. This would reduce the requirements from 8k per second to as little as 3k.

Even so, the final bottleneck was often the file system itself. File systems have complex meta structures, including index blocks that need to be updated when files expand. These introduce additional block seeks and additional latencies to the process of recording an audio file. To get around this problem completely, early voice processing systems would often use a special partition organized as disk blocks, with a very simple meta-structure to maintain housekeeping. One would then perform physical I/O directly to known disk blocks rather than through a file system, and hence early hardware would also provide audio in chunks that were typically aligned to disk blocks, such as in 4096 or 16384 byte chunks.

So to write audio to disk, one would setup a time constrained realtime process that would take a block of encoded audio, and write it to physical blocks on a storage system. As systems grew in complexity and were asked to do other tasks as well, the need for priority scheduling to assure these disk i/o requests would also be completed became very important. While many early voice processing systems were written as custom systems, commercial realtime systems, such as QNX, where often used for building automated telephony applications because they could offer deterministic scheduling with very low latency and also act as a generic platform one could support non-time constrained tasks on top of.

While not discussed here, the act of playing audio samples that had been recorded is also a time constrained process. However, playing audio often has greater flexibility. One can pause audio playback, at least if one is not in the middle of an utterance. Also, one does not need to locate new index blocks when playing audio from the file system. So, the process of playing, while having many of the same constraints, often is less challenging to the system design than that of recording.

Today the mere act of writing audio samples to disk is not such a great challenge. Many modern voice processing systems now use the luxury of writing to the file system rather than trying to optimize raw I/O operations. However, the time constrained nature of recording human speech still exists. While hard realtime systems are no longer necessary, it can still be desirable to have deterministic scheduling for such a task. This can be achieved within the limited goals of soft-realtime that are offered to process scheduling in the modern Linux kernel when applied correctly to this problem. Today, doing so is more a matter of overall system performance tuning rather than a hard requirement.

Today, the same server that might be running a voice application is likely to be doing many other things, including running a web server, running java, etc, and each of these other tasks have their own effect on system loading and performance. Yet, recording voice is still a time constrained activity. I think the best way of handling such systems is to divide processes between those that have realtime constraints and those that do not. Certainly a web server has no realtime constraints, but it's use of a java serverlet can draw down system load excessively. Realtime scheduling then must assure that a realtime process with repeated interval execution requirements is able to gain cpu access within that time constraint, and that it is able to complete i/o operations in a limited time to assure we do not hold the rest of the system hostage.

3 I/O Bottlenecks and Priority Inversion

The Linux kernel extensions for soft realtime do allow one to schedule and prioritize realtime tasks differently from user tasks, but the question of i/o completion is separate and best addressed in a true async i/o subsystem. This would allow such a tasks to perform a request but not have to wait for it's completion, or stutter the system due to a drive defect or filesystem quirk. This suggests there are still interesting engineering problems to consider even in the simple act of recording audio, even if one does not have to consider the entire system design to be constrained by it as had to be the case in the past.

The reason for asynchronous disk I/O is related to blocking. Very often, we do not want a telephony server to block. Blocking could mean a stall on all voice processing on all channels, depending on the nature of the block. For example, if the server blocks because a disk block write has failed and the drive is undergoing some internal retry, we do not want the current task to wait for the block write to complete before returning. If it does, and other tasks are scheduled that also want to use the drive, each of those tasks then become dependent on the completion of the current tasks that had a failed disk write.

While most priority inversion scenarios are understood as occurring through IPC, priority inversion can also in effect occur if the device (such as the harddisk) is blocked on an incomplete I/O operation that then also prevents even higher priority tasks which need the same device from operating because they also become blocked. This could allow an even realtime telephony audio recording service to potentially be held hostage to any other running process that may issue a disk request.

Asynchronous I/O can eliminate device contention and priority inversion due to device contention. As such, I see asynchronous I/O services as a compliment to, if not a requirement for, user mode realtime services.

4 Packetized Voice

With the widespread introduction of packetized voice over Ethernet, a whole new range of issues have come into existence related to latency and these offer new realtime challenges that a modern kernel must successfully address. While I am mostly speaking here about packetized voice carried over ethernet, the same fundamental issues apply to wireless voice telephony networks.

To understand packetized voice transport, one must consider that a frame of voice has to be sampled. Each frame is then sent or streamed to a remote host. This is done using UDP because TCP includes flow control. When people talk, they do not have underlying flow control in their intercommunication medium. Packetized voice also deals with conversations, and hence are full duplex, where each side is both sending and receiving audio.

Since UDP transport is commonly used, packets can arrive out of order, and some may be entirely missing. This means that one cannot simply play packets as they arrive. There has to be enough time to assure that the next packet will arrive and to see if when packets are out of order, if the missing packet may arrive, before playing can continue. Voice must be played continually, and so often a temporal receive queue is established with a time window for additional packets to arrive. This means there is a delay between when packets arrive and when they are played, to allow sufficient time for missing or mis-ordered packets to arrive.

Playing itself is of course a realtime process, and if a packet is still missing even with a time delay offset, it has to be ignored, or filled with a blank packet, and if it arrives later it is ignored. Otherwise, if one waits for a missing packet (that may in any case never arrive considering UDP), then the far end node will be time delayed further from the original speaker. In a one way conversation this may be acceptable, but in full duplex conversations this does not work.

What has been found is that people can converse full duplex with some delay between transmission and reception without being disturbed so long as the total delay is under 250 milliseconds. When one approaches or exceeds this limit, then conversation becomes awkward and difficult.

To packetize voice, one has to sample a buffer of a known size, and then transmit it. The buffer that is sampled in realtime is delayed during the sampling. The size of a frame must be sufficiently large that the data portion of the packet is not small compared to the IP header otherwise bandwidth is being waisted. The frame size cannot be so large that it uses up a significant amount of that 250ms total time window, however. Since additional processing on each frame is needed, there are other constraints on size as well.

For example, one might need to run dsp algorithms on a frame that is being packetized. This might be done to detect energy level to see if a frame is mostly silent. Silent frames do not need to be sent, thereby saving additional bandwidth.

We may also need to extract telephony tones. In particular, we may need to detect and extract DTMF from the sample frame. While dsp's work well on fixed size frames, there are a minimal number of waveforms needed to be able to algorithmically detect the presence of a given frequency tone, and this also imposes some limitations.

Generally, when all these and other considerations are taken into account, voice is packetized in 10, 20, or 30ms frame increments, with 20ms being the most common for IP voice systems. Hence, every 20ms the transmitter will be transmitting a frame to a receiver. At most one might delay 10ms over that. Clearly the act of sampling and transmitting audio frames is a time realtime user space task, with tight time constraints and a low latency requirement. If many concurrent conversations are occurring through the same machine, then many realtime tasks need to be handled, each of equal priority.

While VoIP systems exist today that run on non-realtime systems, many of these systems are only expected to carry on a single conversation. A PC desktop softphone client, for example, will likely only hold one conversation at a time. The shear muscle of todays systems can brute force such limited needs. I believe hard realtime scheduling becomes important when considering server designs, where a server may handle upward of several hundred simultaneous time constrained packetized voice sessions. Many systems do this today through the liberal use of external hardware that typically runs a realtime kernel and offloads the hard parts from the host machine, and this I will get back to a bit later.

5 Host DSP processing

In the past there had been a trend to make computer telephony hardware more complex. This was often done to remove the realtime constrains away from the PC platform, especially with the prevalence of low performance, non realtime capable, or unreliable commodity proprietary operating systems. These systems pushed hardware designs to the point that often such hardware would provide it's own extensive buffering and framing of audio. With the rise of DSP chips, many of these cards began migrating functions that used to require additional hardware directly into DSP firmware.

Functions that traditionally were done in hardware include encoding and decoding of voice through different codec's. Hardware codec's were typically limited to things like ADPCM encoding. With DSP's, one can apply any codec, including codec's that operate on differently sized frames, such as mp3 audio, rather than on simple bit level transforms the way ADPCM operates.

Another very basic telephony function is tone detection. Tone detection is often associated with being able to listen for DTMF digits from telephone keypads. Tone detection may also be used to detect the presence of dialtone, or call progress tones such as the busy tone or central office intercept tones. Detection of tones, while originally involving discreet logic chips, have also migrated to more general purpose DSP firmware.

FAX tones and fax processing, while not purely realtime, and conceptually simple, involves handshaking, and handshaking based on content analysis was often beyond the capabilities of simple electronic circuits. While chips were created to do fax, for most modern computer telephony hardware, fax is also often handled through dsp processing.

As DSP chips became more powerful, many additional tasks have been offloaded to them. Some computer telephony hardware offloads speech synthesis and speech recognition to DSP hardware, for example. Other tasks can include biometric voice profiling.

As ever more complex and powerful DSP chips are used in computer telephony hardware, the costs of such hardware has remained high. This is often because DSP resources are needed that are sufficient to handle dozens, or even hundreds, of simultaneous telephone sessions. Also, many of the dsp firmware algorithms are licensed in a proprietary manner, and some DSP vendors claim patents over specific DSP algorithms today.

This high cost has lead some hardware designers to experiment with host based DSP processing, using the CPU resources of the server, rather than a dedicated DSP chip. This was first seen in the introduction of things like the so called

”winmodems”. Host based processing is often difficult on an operating system with low task latency and non-realtime scheduling capabilities, and certainly computer telephony hardware using host processing would have been primarily limited to single port devices if those were the only choice of commodity operating system platforms available.

Host based processing often occurs at the kernel level. This is demonstrated by Voicetronix and Digium. Both of these vendors produce multi-port telephony hardware for interfacing to the public telephone network for use with GNU/Linux systems with no on-board DSP. Both have chosen to make use of kernel modules to embed dsp functionality.

I am not familiar with Digium’s hardware. However, at least in the case of Voicetronix, what they do is create a somewhat more realtime kernel environment by daisy chaining a hardware interrupt and using that to schedule their host dsp processing code. This code of course offers no general solution and is highly dependent on the kernel environment itself as it changes.

Since Voicetronix processes full duplex conversations, and people cannot be flow controlled, the ability to process voice frames in a realtime manner is critical. In fact, to support this, they choose to use a 1ms buffer, and to copy audio between source and destination ports directly when supporting a conference. The reason for the short buffer was to eliminate the need for computationally intensive echo cancellation.

That this works at all, and works well for 12 or more simultaneous voice calls, is that the Linux kernel, as well designed as it is, has a very low interrupt response latency. Latency of course is an important benchmark of realtime performance. Deterministic scheduling, as noted, is achieved through a pure kernel ISR driven by hardware.

While crude, what we learn from the Voicetronix experience is that there is a need for a more general and abstract kernel level support for realtime scheduling, as well as realtime support at the user process level. There are in fact some realtime Linux projects that either do that or run the Linux kernel itself on top of a simple realtime core kernel. These projects span a large domain of different efforts, such as the l5-linux project, the RT-Linux kernels, etc. It would seem useful for one of these to become adopted as a standard feature that one can find in a pre-packaged GNU/Linux distribution, as only then will the creation of hacks like what Voicetronix and Digium are no doubt doing be eliminated by the introduction of standard practices.

6 Embedded kernels for telephony hardware

As I noted earlier, many computer telephony vendors have chosen to move realtime processing off the host and onto dedicated hardware. This is particularly true of high port capacity IP voice telephony cards. These cards exist in capacities large enough to service a DS3 or more worth of voice traffic (700-2000 simultaneous conversations). Since each and every IP voice frame has to be analyzed for DTMF tones, they make extensive use of DSP firmware. But they also provide packetized realtime UDP sessions, and these sessions are typically managed with an embedded custom realtime operating system kernel.

The linux kernel is certainly light enough to be successfully embedded into a card or other small footprint device. This particular kind of application, while depending on standard networking protocols, would also emphasize the need for hard realtime services. At any time, maybe 1000 separate 20ms frames will need to be gathered, handed off to a DSP, and passed through a send queue. Similarly, the receiver would have to provide a new audio frame from it’s receive queue every 20ms. This application requires no disk drivers and no use of blocking resources. But it does require low context switch latency, and true deterministic scheduling.

7 Putting the pieces together into GNU Bayonne

With the realization that GNU/Linux systems today could be effectivily used to create telephony application services, we set out to create GNU Bayonne. GNU Bayonne is a middleware telephony server that can be used to create and deploy script driven telephony application services. These services interact with users over the public telephone network. Using commodity PC hardware and CTI cards running under GNU/Linux available from numerous vendors, GNU Bayonne can be used to create carrier applications like Voice Mail and calling card systems, as well as enterprise applications such as unified messaging. It can be used to provide voice response for e-commerce systems and has been used in this role in various e-gov projects. GNU Bayonne can also be used to telephony enable existing scripting languages such as perl and

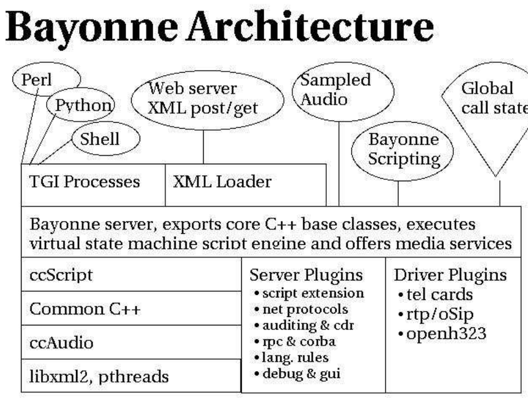

Figure 1: Architecture of GNU Bayonne

python.

GNU Bayonne has to interact with telephony devices and many concurrent users, and deal with the potential realtime requirements that this involves. At the same time, it has to be able to provide application logic, which, while potentially computativily intensive, or, more often, disk intensive, such as when performing a database query, is generally not an activity scheduled on a realtime or time constrained basis.

Furthermore, while the functional requirements are actually fairly simple, each vendor of computer telephony hardware has chosen to create their own unique and substantial application library interface, we needed GNU Bayonne to sit above these and be able to abstract them. Ultimately we choose to create a driver plugin architecture to do this. What this means is that you can get a card and api from Aculab, for example, write your application in GNU Bayonne using it, and later choose, say, to use Intel telephony hardware, and still have your application run, unmodified. This has never been done in the industry widely because many of these same telephony hardware manufacturers like to produce their own middleware solutions that lock users into their products.

These differing roles and requirements lend themselves to a somewhat complex architecture, as can be seen here:

As can be seen, we bring all these elements together into a GNU Bayonne server, which then executes as a single core image. The server itself exports a series of base classes which are then derived in plugins. In this way, the core server itself acts as a “library” as well as a system image. What is unique about this is that When the server comes up, it creates new objects by loading plugins. The plugins themselves use base classes found in the server and derived objects that are defined for static storage. This means when the plugin object is mapped thru dload, it’s constructor is immediately executed, and the object’s base class found in the server image registers the object with the rest of GNU Bayonne. Using this method, plugins in effect automatically register themselves thru the server as they are loaded, rather than thru a separate runtime operation.

To couple the realtime requirements of the telephony world with lazier application logic, two approaches are found in Bayonne. First, a state machine represents the operation of the abstracted driver interface. This state machine is executed in a non-blocking manner over multiple threads through event callback, and in conjunction with a non-blocking and step driven script system known as GNU ccScript. GNU ccScript statements either execute and return immediately, or they schedule their completion for a later time with the GNU bayonne state machine executive. This allows a single thread to efficiently invoke and manage multiple interpreter instances. While GNU Bayonne can support interacting with hundreds of simultaneous telephone callers on high density carrier scale hardware, we do not require hundreds of native ”thread” instances running in the server, and we have a very modest cpu load.

Another way GNU ccScript is unique is in support for memory loaded scripts. To avoid delay or blocking while loading scripts, all scripts are loaded and parsed into a virtual machine structure in memory. When we wish to change scripts, a brand new virtual machine instance is created to contain these scripts. Calls currently in progress continue under the’ old vm and new callers are offered the new vm. When the last old call terminates, the entire old vm is then disposed of. This allows for 100uptime even while services are modified.

The constrained nature of GNU ccScript does not nessisarly allow it’s use as a complete telephony server application solution. It cannot communicate with databases directly as these operations can block, as in one example of it’s limitations. While GNU Bayonne’s server scripting can support the creation of complete telephony applications, it was not designed to be a general purpose programming language or to integrate with external libraries the way traditional languages do. The requirement for non-blocking requires any module extensions created for GNU Bayonne are written highly custom. We wanted a more general purpose way to create script extensions that could interact with databases or other system resources, and we choose a model essentially similar to how a web server does this.

The TGI model for GNU Bayonne is very similar to how CGI works for a web server. In TGI, a separate process is started, and it is passed information on the phone caller thru environment variables. Environment variables are used rather than command line arguments to prevent snooping of transactions that might include things like credit card information and which might be visible to a simple “ps” command.

The TGI process is tethered to GNU Bayonne thru stdout and any output the TGI application generates is used to invoke server commands. These commands can do things like set return values, such as the result of a database lookup, or they can do things like invoke new sessions to perform outbound dialing. A “pool” of available processes are maintained for TGI gateways so that it can be treated as a restricted resource, rather than creating a gateway for each concurrent call session. It is assumed gateway execution time represents a small percentage of total call time, so it is efficient to maintain a small process pool always available for quick TGI startup and desirable to prevent stampeding if say all the callers hit a TGI at the exact same moment.

This ability to mix realtime and non-realtime capabilities and processing into a comprehensive whole is what I find most useful in the Linux kernel today. This is a key feature needed to create telephony applications, whether we speak about GNU Bayonne or any other advanced telephony server.

8 Conclusion

GNU Bayonne is of course just one example of how linux kernel realtime capabilities and choices can effect application design and implimentation of telephony services under GNU/Linux. The Linux kernel has many roles it can successfully play in the telecommunications marketplace. As an application services platform, the Linux kernel, even with limited soft realtime and asynchronous I/O, can meet the needs of today’s telephony applications better very well. With hard realtime extensions, including guaranteed scheduling, the Linux kernel could be used in many embedded telecommunication devices, and with improved support for kernel level realtime capabilities, it may permit the deployment of commodity telephony servers with ever more inexpensive computer telephony hardware.