GNU Bayonne: telephony application server of the GNU project

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

Abstract

GNU Bayonne is a freely licensed middle-ware telephony server that can be used to create and deploy application script driven telephony application services. These services interact with users automatically 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, and even complete stand-alone commercial telephone systems. 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 python.

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 been the domain of 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 real-time 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.

In addition to real-time constraints, telephony servers are usually housed in phone closets or other closed and isolated areas. As such, remote maintainability, and high reliability are both important platform requirements as well. The ability to integrate with and use standard networking protocols is also becoming very important in traditional telephony, and certainly is a key requirement for next generation telephony platforms.

So we can summarize; low latency/high performance kernels, remote manageability without the need for a desktop environment, high reliability, and open networking protocols. This sounds like an ideal match for a GNU/Linux system. For these reason we choose to build telephony services such as GNU Bayonne primarily under GNU/Linux.

Our goal for GNU Bayonne 1.0 was primarily to make telephony services as easy to program and deploy as a web server is today. We choose to make this server easily programmable through server scripting. We also desired to have it highly portable, and allow it to integrate with existing application scripting tools so that one could leverage not just the core server but the entire platform to deliver telephony functionality and integrate with other resources like databases.

Since each vendor of telephony hardware has chosen to create their own unique and substantial application library interface, we faced another key issue in designing a generic telephony middleware server. We needed GNU Bayonne to sit above these and be able to abstract them. Ultimately we choose to create a driver plug-in 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 middle-ware solutions that lock users into their products. This ability to abstract and replace hardware without changing application services also became a key requirement for Bayonne.

2 Real-time constraints

GNU Bayonne, as a telephony server, imposes some very real and unique design constraints. For example, we must provide interactive voice response in real-time. “real-time” in this case may mean what a person might tolerate, or delay of 1/10th of a second, rather than what one might measure in milliseconds in other kinds of real-time applications. However, this still means that the service cannot block, for, after all, you cannot flow control people speaking.

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 real-time 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 real-time 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 real-time 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.

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 real-time 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-real-time that are offered to process scheduling in the modern Linux kernel when applied correctly to this problem, and enables us to use a standard GNU Linux system to run GNU Bayonne even for very large capacity systems. Today, doing so is more a matter of overall system performance tuning rather than a hard requirement.

3 C++ core development

To create GNU Bayonne we needed a portable foundation written in C++. I wanted to use C++ for several reasons. First, the highly abstract nature of the driver interfaces seemed very natural to use class encapsulation for. Second, I found I personally could write C++ code faster and more bug free than I could write C code.

Why we choose not to use an existing framework is also simple to explain. We knew we needed threading, and socket support, and a few other things. There were no single framework that did all these things except a few that were very large and complex which did far more than we needed. We wanted a small footprint for Bayonne, and the most adaptable framework that we found at the time typically added several megabyte of core image just for the runtime library.

GNU Common C++ (originally APE) was created to provide a very easy to comprehend and portable class abstraction for threads, sockets, semaphores, exceptions, etc. This has since grown into it's own and is now used as a foundation of a number of projects as well as being a part of GNU.

4 C++ scripting engine

In addition to having portable C++ threading, we needed a scripting engine. This scripting system had to operate in conjunction with a non-blocking state-transition call processing system. It also had to offer immediate call response, and support several hundred to a thousand instances running concurrently in one server image.

Many extension languages assume a separate execution instance (thread or process) for each interpreter instance. These were unsuitable. Many extension languages assume expression parsing with non-deterministic run time. An expression could invoke recursive functions or entire sub-programs for example. Again, since we wanted not to have a separate execution instance for each interpreter instance, and have each instance respond to the leading edge of an event callback from the telephony driver as it steps through a state machine, none of the existing common solutions like tcl, perl, guile, etc, would immediately work for us. Instead, we created a non-blocking and deterministic scripting engine, GNU ccScript.

GNU ccScript is unique in several ways. It is step executed, and is non-blocking. Statements either execute and return immediately, or they schedule their completion for a later time with the executive. A given ”step” is executed, rather than linearly. This allows a single thread to 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 virtual machine and new callers are offered the new virtual machine. When the last old call terminates, the entire old virtual machine is then disposed of. This allows for 100% uptime even while services are modified.

Finally, GNU ccScript allows direct class extension of the script interpreter. This allows one to easily create a derived dialect specific to a given application, or even specific to a given GNU Bayonne driver, simply by deriving it from the core language through standard C++ class extension.

5 TGI support and plug-ins

To be able to create useful applications, it is necessary to have more than just a scripting language. It requires a means to be extended so that it can incorporate database access libraries or other functions that fall outside of the scope of the scripting language itself. These extensions should be loaded on demand only when used, and should be specified at runtime so that new ones can easily be added without the need to recompile the entire server.

To support scripting extensions we have the ability to create direct command extensions to the native GNU Bayonne scripting languages. These command extensions can be processed through plug-in modules which can be loaded at runtime, and offer both scripting language visible interface extensions, and, within the plug-in, the logic necessary to support the operation being represented to the scripting system. These are much more tightly coupled to the internal virtual machine environment and a well written plug-in could make use of thread pools or other resources in a very efficient manner for high port capacity applications.

When writing command extensions, it is necessary to consider the need for non-blocking operations. GNU Bayonne uses ccScript principally to assure non-blocking scripting, and so any plug-in must be written so that if it must block, it does so by scheduling a state operation such as ”sleep” and performs potentially blocking operations in separate threads. This makes it both hard and complex to correctly create script extensions in this manner.

The constrained nature of GNU ccScript does not fully 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.

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 through 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 through 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.

6 Bayonne Architecture

With the realization that GNU/Linux systems today could be effectively used to create telephony application services, we set out to create GNU Bayonne, and came up with a common architecture to define the operations of a telephony middleware server. As noted earlier, GNU Bayonne had 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, as noted earlier, each vendor of computer telephony hardware has chosen to create their own unique and substantial application library interface, thus requiring extreme abstraction between the lower level telephony driver or api and the Bayonne application environment. 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 plug-ins. In this way, the core server itself acts as a “library” as well as a system image. What is unique

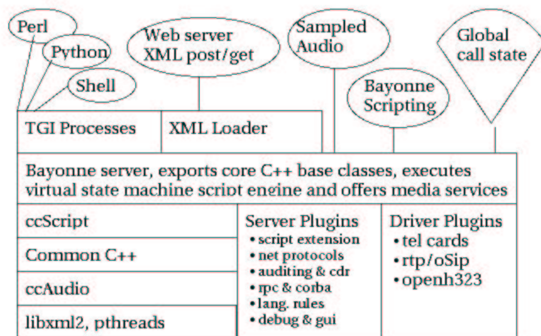

Figure 1: Architecture of GNU Bayonne

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 GNU ccScript, which, as noted earlier, behaves as a non-blocking and step driven script system, which is then executed directly from the state machine driver.

Since GNU Bayonne has to interact with telephone users over the public telephone network or private branch exchange, there must be hardware used to interconnect GNU Bayonne to the telephone network. There are many vendors that supply this kind of hardware and often as PC add-on cards. Some of these cards are single line telephony devices such as the Quicknet LineJack card, and others might support multiple T1 spans. Some of these cards have extensive on-board DSP resources and TDM buses to allow interconnection and switching.

GNU Bayonne tries to abstract the hardware as much as possible and supports a very broad range of hardware already. GNU Bayonne offers support for /dev/phone Linux kernel telephony cards such as the Quicknet LineJack, for multiport analog DSP cards from VoiceTronix and Dialogic, and digital telephony cards including CAPI 2.0 (CAPI4Linux) compliant cards, and digital span cards from Intel/Dialogic and Aculab. We are always looking to broaden this range of card support.

At present both voice modem and OpenH323 support is being worked on. Voice modem support will allow one to use generic low cost voice modems as a GNU Bayonne telephony resource. The openh323 driver will actually require no hardware but will enable GNU Bayonne to be used as an application server for telephone networks and softswitch equipment built around the h323 protocol family. At the time of this writing I am not sure if either or both of these will be completed in time for the next stable (1.2) release.

7 Current Status

The 1.0 release of GNU Bayonne was distributed in September of 2002. This release represented several years of active development and has standardized how GNU Bayonne operates and is deployed. In November, we choose to make a 1.1 release available which demonstrated the ability for GNU Bayonne to operate as a complete small office telephone system. Current development today includes the ability to embed sql and database access services into Bayonne, and to improve it's usability as a platform for telephony enabling web services, as well as work on supporting next generation telephone networks with GNU Bayonne.

GNU Bayonne does not exist alone but is part of a larger meta-project, "GNUCOMM". The goals of GNUCOMM is to provide telephony services for both current and next generation telephone networks using freely licensed software. These services could be defined as services that interact with desktop users such as address books that can dial phones and softphone applications, services for telephone switching such as the IPSwitch GNU softswitch project and GNU oSIP proxy registrar, services for gateways between current and next generation telephone networks such as troll and proxies between firewalled telephone networks such as Ogre, real-time database transaction systems like preViking Infotel and BayonneDB, and voice application services such as those delivered through GNU Bayonne.

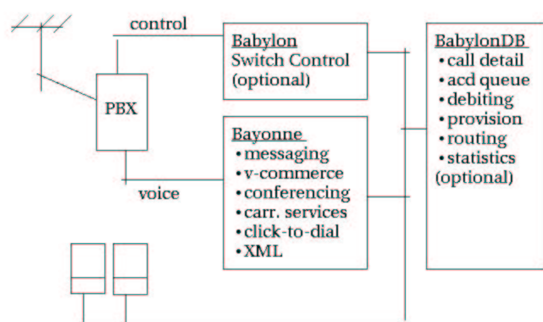

Figure 2: Enterprise Applications

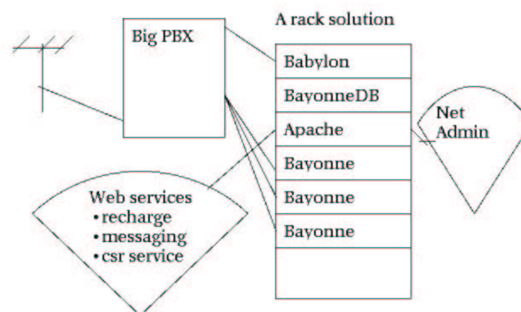

Figure 3: Carrier Applications

8 Enterprise and Carrier Applications

GNU Bayonne 1.0, with the help of other components being developed as part of GNUCOMM, does enable one to create and deploy scalable enterprise and carrier class applications using GNU/Linux systems.

In our broadest view of enterprise telephony applications, we can see using GNU Bayonne as providing a part of overall enterprise telephony solutions. GNU Bayonne must be able to interact with enterprise data, whether through transaction monitors such as BayonneDB, with native sql capability, through web services and XML document processing, or through perl scripts executed via TGI. It may need to interact with other services such as email when delivering voice messages to a unified mailbox, or the local phone switch through integration resources and other kinds of servers.

Our view of GNU Bayonne and telephony application services are that it is a strategic and integral part of the commercial enterprise. Proprietary solutions that are in common use today have often been designed from the question of how to lock a user into a specific OEM product family and control what a user or reseller can do or integrate such products, rather than from the question of what the enterprise user needs and how to provide the means to enable it. This has often kept telephony separate and walled off from the rest of the enterprise. We do not wish to see it separate but a natural extension, whether of web services, of contact management, of customer relations, etc.

When we look at carrier class applications for GNU Bayonne today, we typically consider applications like operator assistance systems, prepaid calling platforms, and service provider voice mail. Each of these has different requirements. What they have in common is that a front end central office switch might be used, such as a Lucent Excel or even a full ESS 5 switch. Application logic and control for voice processing would then be hosted on one or more racks of GNU Bayonne servers most likely interconnected with multiple T1 spans. If database transactions are involved, such as in pre-paid calling, perhaps we would distribute a BayonneDB server to provide database connectivity for each rack. A web server may also exist if there is some web service component.

Operator assist services are probably the easiest to understand. Very often a carrier might need to provide directory assistance or some other form of specialized assist service. A call will come in from the switching center to a GNU Bayonne server, which will then decide what to do with the call. If the caller is from a location that is known, perhaps the call will be re-routed by GNU Bayonne through an outgoing span to a local service center. Online operator assistance might be done by creating an outgoing session to locate an operator and then bridge the callers, all on a GNU Bayonne server.

In service provider voice mail one doesn't have to bridge calls. Service provider voice mail is typically much simpler than enterprise voice mail; there is no company voice directory, there is no forwarding or replying between voice mailboxes, there may be no external message notification. All these things make it an easy to define application on first appearance. What it must be is reliable, and ideally scalable.

Many applications carriers wish to deploy do not necessary

require “carrier grade” Linux to appear before they can be used. In fact, IDT Corp, a major provider of prepaid calling in the world today, uses over 500 rack mounted commodity PC’s running things including a standard distribution of “RedHat” GNU/Linux to reliably service over 20 million call minutes per day in their main switching center. This does not mean there is no value in the carrier grade kernel work, just that it is not necessary to create and sell some types of GNU/Linux voice processing solutions for carriers today. We have looked at the issues involved in high reliability/carrier grade enhanced Linux and we intend to address those as described a little further.

9 GNU Bayonne and web services

Some people have chosen to create telephony services through web scripting, which is an admirable ambition. To do this, several XML dialects have been created, but the idea is essentially the same. A query is made, typically to a web server, which then does some local processing and spits back a well formed XML document, which can then be used as a script to interact with the telephone user. These make use of XML to generate application logic and control much like a scripting language, and, perhaps, is an inappropriate use of XML, which really is designed for document presentation and inter- exchange rather than as a scripting tool. However, given the popularity of creating services in this manner, we do support them in GNU Bayonne.

GNU Bayonne did not choose to be designed with a single or specific XML dialect in mind, and as such it uses a plug-in. The design is implemented by dynamically transcoding an XML document that has been fetched into the internal ccScript virtual machine instructions, and then execute the transcoded script as if it were a native ccScript application. This allows us to transcode different XML dialects and run them on GNU Bayonne, or even support multiple dialects at once.

Since we now learn that several companies are trying to force through XML voice browsing standards which they have patent claims in, it seems fortunate that we neither depend on XML scripting nor are restricted to a specific dialect at this time. My main concern is if the W3C will standardize voice browsing itself only to later find out that the very process of presenting a document in XML encoded scripting to a telephone user may turn out to have a submarine patent, rather than just the specific attempts to patent parts of the existing W3C voice browsing standard efforts.

In addition to being able to process information from web sites, we had wished to enable GNU Bayonne itself to be invoked from other web services. To achieve this, we have been working on RPC services, particularly based on ”soap”, to enable web services to make RPC requests to a Bayonne server. These requests can take several forms.

Some requests can be serviced immediately. These are typically administration requests. Such requests can be basic things like ”stop” and ”start” the server, or more complex things such as setting and querying real-time server information.

In addition to manipulating the server through a web service, it is also desirable to manipulate Bayonne applications. For example, if one has created a web service, such as phpgroupware, which has an online address book, it might be desirable to use a Bayonne server to make and complete telephone calls on behalf of address book entries. These services do not have an immediate return because they depend on the result of Bayonne running a script application which itself may have an undetermined runtime and uncertain results.

To support web services that must invoke Bayonne to run applications, we have enabled two rpc features. The first is a transaction log. A Bayonne script can post information to a transaction log as it is running. This could be basic success or failure information, or even extended error codes. The second is the introduction of an rpc method to retrieve the transaction log entry so that the external web service can determine what happened to the application it initiated through Bayonne. While this method requires some polling to retrieve results, it is not believed that the kind of applications launched require much immediate response on the part of the web service, and in fact, the web service could often choose to examine the log result of a call session long after the application has terminated rather than continually polling.

This interaction allows Bayonne to be integrated both as a middleware telephony resource for other web services, and as a consumer of web services published information.

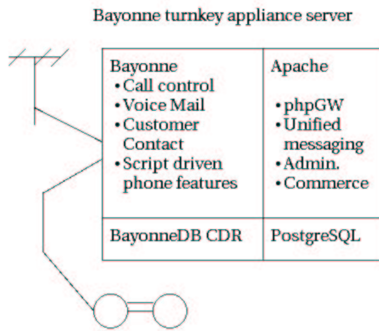

Figure 4: GNU Bayonne Phone Systems

10 A Bayonne Telephone System

Throughout the entire development of GNU Bayonne, it had been used as a middleware adjunct platform that would sit behind a telephone switch or circuits provided by a commercial carrier. When applications needed to interact with local users, this was often done by call transfer operations initiated through the telephone switch or using Centrex-like services.

With the availability of computer telephony hardware which would allow one or more analog telephone to be plugged directly into such cards, and with drivers for this hardware available on GNU/Linux systems, we looked at the feasibility of extending Bayonne's ability to abstract telephony hardware to the question of abstracting hardware that also had telephones directly attached. This development was initiated over a period of two months, and resulted in the first 1.1 release of GNU Bayonne in early November.

In having a complete phone system under GNU Bayonne, it is possible to have a single platform that provides both dialtone and application services. Traditionally these had been split among multiple boxes, and hence had a higher cost of ownership even before the question of software cost and proprietary licensing issues are considered.

We choose to use GNU Bayonne's script driven channel interface to support telephone extensions in much the same manner that trunk ports are used. This means that the feature set of a given telephone station can be programmed and controlled directly through scripting. This allows one to simulate common features found in other telephone systems, and to create or prototype entirely new features from scratch. This also means that telephone features can be modified as needed to create highly integrated systems to meet the needs of vertical markets much more easily than today's typical general purpose phone systems can do.

An example of meeting a vertical market need though application scripting would be a hotel system. Imagine a hotel system where, when one tries to dial out, the guest is interactively prompted to confirm phone activation for billing purposes, rather than having to call down to the front desk. While this level of integration can be achieved in some highly customized systems developed for target market uses that are available today, GNU Bayonne can do this in a completely commodity fashion on a non-custom platform simply by constructing appropriate customized application scripting.

Supporting a complete phone system in GNU Bayonne did require a whole bunch of new functions to be created. These include functions to synchronize connections, to pickup and drop lines, to provide intercom dialing, and to ring stations. The range of common telephone system features that have been created and demonstrated to date in GNU Bayonne 1.1 include intercom dialing, call park, automatic and announced call transfer, call recall, call pickup, hunting, and call coverage. Other features can be created through further scripting.

Since all the voice processing capabilities of Bayonne are available to be applied through scripting, it is possible to impliment voice processing applications directly on a Bayonne telephone system. This can be simple things, like a voice mail system that announces to callers they have messages waiting when they pick up the phone, or very complex ivr and acd type systems, with Bayonne acting as a universal middleware platform. In addition, we are looking to integrate desktop functionality with GNU Bayonne telephone switches, so that one can use address books and have them dial contacts, and receive screen pops for incoming calls. In the end we are looking to create a complete freely licensed turnkey system any commercial entity can use for a complete telephone server.

11 Bayonne and IP Voice networks

While Bayonne has traditionally been used to provide telephony applications that operate with hardware that interfaces to the public telephone network, it was recognized fairly early on that one can abstract IP voice networks and protocol stacks

to Bayonne as if they were physical telephone circuits integrated through multiport channel cards. This has lead to the idea of providing plugin driver interfaces to enable GNU Bayonne to operate as an application server in an IP voice network.

One curious advantage to be found in using Bayonne this way is application reusability. Bayonne applications are generally not aware of the underlying telephony driver, and so one can create applications that are prototyped on single analog circuits and apply them unmodified to high density digital spans. Similarly, one would be able to take existing Bayonne applications developed for the wired telephone network and re-use them unmodified in an IP voice network. This also means a single middleware platform could exist for creating and deploying applications for both current and next generate telephone networks.

Early on, development has proceeded in two paths. The first path was to integrate the existing OpenH323 project to provide H323 call control endpoints and sessions for running Bayonne applications. A second development path was focused on using a RTP stack (ccRTP) for media operations and the GNU oSIP stack for inter-operation of Bayonne with SIP based telephone networks. Both these paths assume Bayonne will abstract RTP sessions as if they were physical trunks and then run existing applications unmodified.

Progress in both these paths have at times been stalled due either to lack of time, resources, or both. I am hoping by the time this paper is available we finally have some demonstratable progress in IP voice networks. I believe this is important for the future of the project, and for the future of developing voice application services. In this environment, since no physical hardware is required, Bayonne's ability to be ported to many different platforms will prove particularly useful. We also view IP voice services as a means to provide all kinds of voice enabled applications over the internet, including automated personal assistants and other things not traditionally seen as part of telephony in the past.

12 Where you can help

While Bayonne has been used commercially in many roles, Bayonne is developed with the help of contributors worldwide. The Bayonne development community is loosly coordinated through a developer mailing list, and through a number of individuals who have taken responsibility for specific functional elements or drivers in Bayonne. We do need more people from additional organizations who can work on the core server features.

In addition, we are looking for individuals and groups to come forward and work on specific Bayonne applications that can be used out of the box. At the moment, I have contributed a simple but complete key telephone system to demonstrate the 1.1 feature set. In addition, we have a project based in Macedonia that is looking for financial sponsorship to develop Bayonne based e-government services for the blind. In fact, we are looking for further commercial or government sponsorship not only for specific sub-projects such as that, but also for supporting continued Bayonne development in general.

There are many areas outside of core development and sponsorship where individuals can often make a difference in Bayonne development. To support portable localization of scripted voice applications and support multi-lingual services, we use a phrase generation system, and we can use contributed voices and help with defining language rules for languages Bayonne does not directly support as yet. Currently Bayonne directlu supports English, French, Italian, Russian, and Bengali. We would like to add many additional languages.

Finally, we are looking for individuals to help in supporting roles in the project. We need people to review and write or improve documentation, and this is a very key role. We also need individuals to come forward and help with maintaining the Bayonne web site, or to simply write more useful Bayonne applications.

13 Conclusion

While we have seen rapid advances in other infrastructure technology, for many decades, progress in telephony has been extremely slow. We believe that replacing the myriad of complex and specialized proprietary telephone systems and supporting platforms today with a common freely licensed middleware that can be universally available and easily customized to meet individual needs will make it possible to bring telephony out of the phone closet and make it possible to rapidly develop telecommunications the way freely licensed software enabled rapid development of the Internet. We also believe that by making freely licensed telephony middleware universally available, it will finally become possible to easily

integrate telephony with and as a natural part of the rest of the computing enterprise.

14 Acknowledgments

There are a number of contributors to GNU Bayonne. These include Matthias Ivers who has provided a lot of good bug fixes and new scheduler code. Matt Benjamin has provided a new and improved TGI tokenizer and worked on Pika outbound dialing code. Wilane Ousmane helped with the French phrasebook rule sets and French language audio prompts. Henry Molina helped with the Spanish phrasebook rule sets and Spanish language audio prompts. Kai Germanschewski wrote the CAPI 2.0 driver for GNU Bayonne, and David Kerry contributed the entire Aculab driver tree. Mark Lipscombe worked extensively on the Dialogic driver tree. There have been many additional people who have contributed to and participated in related projects like GNU Common C++ or who have helped in other ways.