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 middle-ware 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 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. Recent advances in CTI 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.

Telephony services 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 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 is 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.

GNU Bayonne, as a telephony server, also 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.

Since each vendor of 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 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.

2 Supporting Libraries

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.

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 subprograms 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.

3 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.

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 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.

4 Bayonne Architecture

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. One advantage of this scheme is that, unlike a true library, the loaded modules and core server do not need to be relocatable, since only one instance is instantiated in a specific form that is not shared over arbitrary processes.

When the server comes up, it creates gateways and loads plug-ins. The plug-ins themselves use base classes found in the server and derived objects that are defined for

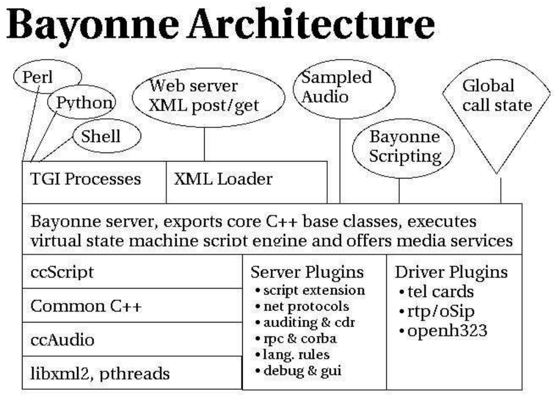

Figure 1: Architecture of GNU Bayonne

static storage. This means when the plug-in object is mapped through 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, plug-ins in effect automatically register themselves through the server as they are loaded, rather than through a separate runtime operation.

The server itself also instantiates some objects at startup even before main() runs. These are typically objects related to plug-in registration or parsing of the configuration file.

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 busses 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 1.0 release.

5 GNU Bayonne and XML Scripting

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.

6 Current Status

Current plans are to distribute the 1.0 release of GNU Bayonne during the opening of this year's LinuxTag, on June 6th. This release represents several years of

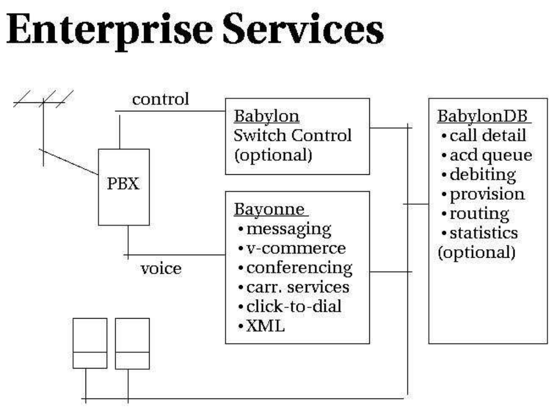

Figure 2: Enterprise Applications

active development and has been standardized in how it operates and how it is deployed. Even before this point, and for the past 6 months, active development has happened on a second generation GNU Bayonne server, and snapshots of this new server are currently available for download. Where GNU Bayonne is evolving will be explained further on.

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, realtime database transaction systems like preViking Infotel and BayonneDB, and voice application services such as those delivered through GNU Bayonne.

7 Enterprise and Carrier Applications

GNU Bayonne 1.0, with the help of other components being developed as part of GNUCOMM, will enable one to create scalable enterprise and carrier class applications under GNU/Linux for the current generation telephone network.

In our broadest view of enterprise telephony applications, we can see using GNU Bayonne as a part of an overall solution. GNU Bayonne must be able to interact with enterprise data, whether through transaction monitors such as BayonneDB or through perl scripts executed via

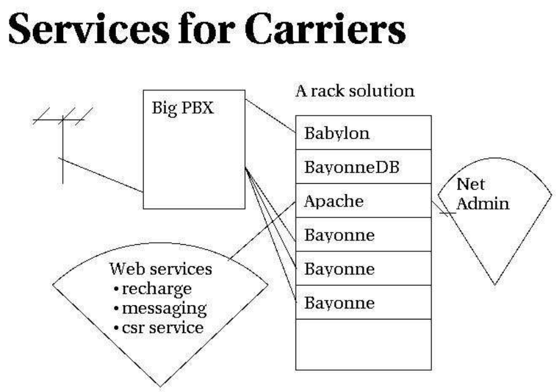

Figure 3: Carrier Applications

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 a resource such as Babylon. We will explain Babylon a bit later.

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.

8 The NG Server

Even before GNU Bayonne 1.0 had been finalized, work had been started by late last year on a successor to GNU Bayonne. This successor attempts to simplify many of the architectural choices that were made early on in the project to make it easier to adept and integrate GNU Bayonne in new ways.

One of the biggest challenges in the current GNU Bayonne server is the creation of telephony card plug-ins. These often involve the implementation of a complete state machine for each and every driver, and very often the code is duplicated. GNU Bayonne "2" solves this by pushing the state machine into the core server and making it fully abstract through C++ class extension. This allows drivers to be simplified, but also enabled us to build multiple servers from a single code base.

Another key difference in GNU Bayonne "2" is much more direct support for carrier grade Linux solutions. In particular, unlike GNU Bayonne, this new server can take ports in and out of service on a live server, and this allows for cards to be hot-plugged or hot swapped. In a carrier grade platform, the kernel will provide notification of changeover events and application services can listen for and respond to these events. GNU Bayonne is designed to support this concept of notification for management of resources it is controlling.

Finally, GNU Bayonne "2" is designed from the ground up to take advantage of XML in various ways. It uses a custom XML dialect for a configuration language. It also acts as a web service with both the ability to request XML content that describe the running state of GNU Bayonne services and the ability to support XMLRPC. This fits into our vision for making telephony servers integrate with web services, represents part of how we envision the project going forward into the future.

9 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.