Automatic Audio Routing for Home Entertainment

Davide Pesavento, Martina Astegno, Claudio E. Palazzi
Università degli Studi di Padova
{dpesaven, mastegno}@studenti.math.unipd.it, cpalazzi@math.unipd.it

***Abstract*—Pervasive technology and communication is improving our life and an interesting case study is represented by domotics. Indeed, innovative technology for home automation can be designed to sense our behavior and proactively respond to it. To this aim, we developed a distributed application exploiting the Bluetooth technology to locate a person wearing a mobile device (e.g., a smartphone) among a predefined set of rooms. This information is then utilized to automatically reroute the output of an audio player from one room to another, according to the person's movements over time.**

***Index Terms*—Audio Routing, Bluetooth, Domotics, PulseAudio**

I. Introduction

Domotics is a science that is gradually expanding and attracting interest from both research and industry. Creating a smart home environment able to exploit automation solutions to serve people represent a challenging task that requires knowledge in different fields of technology. Just mentioning few computer science related topics, an automated home involves the combination of networked appliances, sensors, and wireless technologies [1], [2].

More and more interconnected devices are soon going to invade our homes aiming at increasing comfort, security, safety and better energy management. For instance, several projects aims at revolutionize the way we manage all the appliances in the kitchen (e.g., oven, fridge, coffee machine), proposing solutions to make them interact both with each other and with the Internet [3], [4].

The possibility to monitor people and recognize their behavior has let to the design of eHealth solutions. Indeed, through cameras, ECGs or other sensors it could be possible to keep track of the health status of people inside the house or just to perform fall detection [5], [6].

Other proposed smart home applications regards security and the possibility to detect intrusions or accidents through sensors [7]. Indeed, sensors have been used to this aim for decades; yet, the development and diffusion of wireless technology for communications has lead to a different approach that can allow users to have complete control of their house even remotely, through a smartphone.

In this context, it is particularly interesting to study how current pervasive technology could be used to improve entertainment experience when at home [8]. For instance, users may appreciate an application able to modify the volume of their media depending on certain circumstances. To this aim, we discuss *Conductor* a new entertainment related solution for smart homes we developed. *Conductor* is a distributed application able to reroute the output of an audio player from one room to another, depending on the position of a user. In essence, Bluetooth technology is utilized to locate a person wearing a mobile device (e.g., a smartphone) among a predefined set of rooms; this information is then utilized to automatically reroute the audio output, according to that person's movements over time.

II. Background

A. Counter: Example of a usage scenario

Jessica is a modern housewife who likes listening to music while doing the housework. Her house is quite big though, and with just one hi-fi in the living room, she cannot hear the music loud enough when she is working in some other rooms. This issue is a major nuisance to her listening experience.

Our *Conductor* is exactly what she is looking for: she just needs to install a special kind of speaker in each room and, as long as she keeps her Bluetooth-equipped mobile phone in her pocket, she can freely move from room to room and the music smoothly follows her without interruptions.

B. Short-range device discovery: Bluetooth

Bluetooth is an open specification for a RF-based system that provides the network infrastructure to enable short range wireless communication of data and voice. It includes a hardware component and a software component. The specification also describes usage models and user profiles for these models. Bluetooth supports two kinds of links: *Asynchronous Connectionless* (ACL) links for data transmission and *Synchronous Connection Oriented* (SCO) links for audio/voice transmission. Figure 1 shows the various layers of Bluetooth protocol suite.

Bluetooth hardware uses *Frequency Hopping Spread Spectrum* (FHSS) to avoid any interferences. A Bluetooth channel is divided into time slots each 625 microseconds in length. The devices hop through these timeslots making 1600 hops per second. This trades bandwidth efficiency for reliability, integrity and security.

In our project we adopted BlueZ version 4, the official Linux Bluetooth stack implementation, that provides support for the core Bluetooth layers and protocols [9]. It is flexible, modular and efficient. Current versions of BlueZ consist of many separate modules, for example:

- Bluetooth kernel subsystem core
- L2CAP and SCO audio kernel layers
- RFCOMM, BNEP, HIDP kernel implementations
- HCI UART, USB, PCMCIA and virtual device drivers
- Bluetooth and SDP userspace daemon (`bluetoothd`)
- Configuration and testing utilities
- Protocol decoding and analysis tools

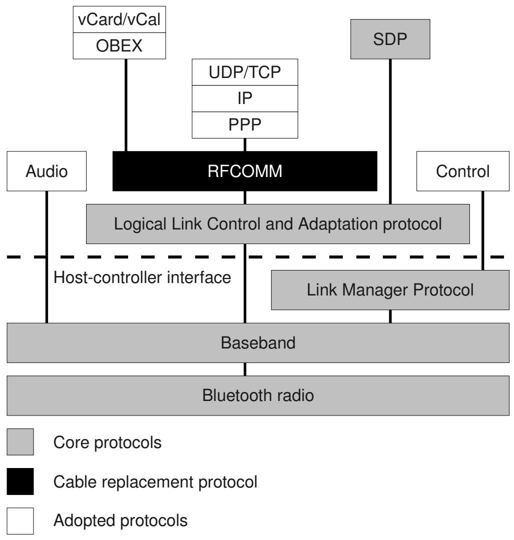

Fig. 1. Bluetooth protocol stack.

C. *Audio routing: PulseAudio*

PulseAudio is a cross-platform, networked sound server [10]. A sound server is basically a proxy for sound applications. It allows to do advanced operations on sound data as it passes between an application and the hardware. Tasks like transferring the audio to a different machine, changing the sample format or channel count and mixing several sounds into one are easily achieved using a sound server. PulseAudio is designed for Linux systems, but has been ported to Microsoft Windows, MacOS X, Solaris, FreeBSD and other POSIX-compliant platforms. Figure 2 depicts a high level overview of PulseAudio architecture.

III. General architecture

The prototype can be divided into 3 primary components, which will be described in greater detail in the next sections, plus a *Qt4* graphical user interface, that will be left out because it does not contain any noteworthy functionalities.

A. *Bluetooth module*

The Bluetooth component deals with the discovery of monitored devices. In particular, the class `ProbeManager` connects to all available Bluetooth adapters upon startup and then sends them appropriate commands to setup or teardown a discovery session.

Since most of these adapters are not directly visible to *Conductor* but they are instead wired up to different machine that is in turn connected to the main application via TCP, the prototype relies on another application, named *Probe*, which is basically a proxy between *Conductor* and the Bluetooth adapters. Main classes and interaction of *Probe* and *Conductor* are shown in Fig. 3. In particular, an instance of *Probe* must be running on each node of the network that has an adapter attached to it. On the whole the application is tiny, efficient and has no GUI, thus it should even be possible to run it on embedded devices.

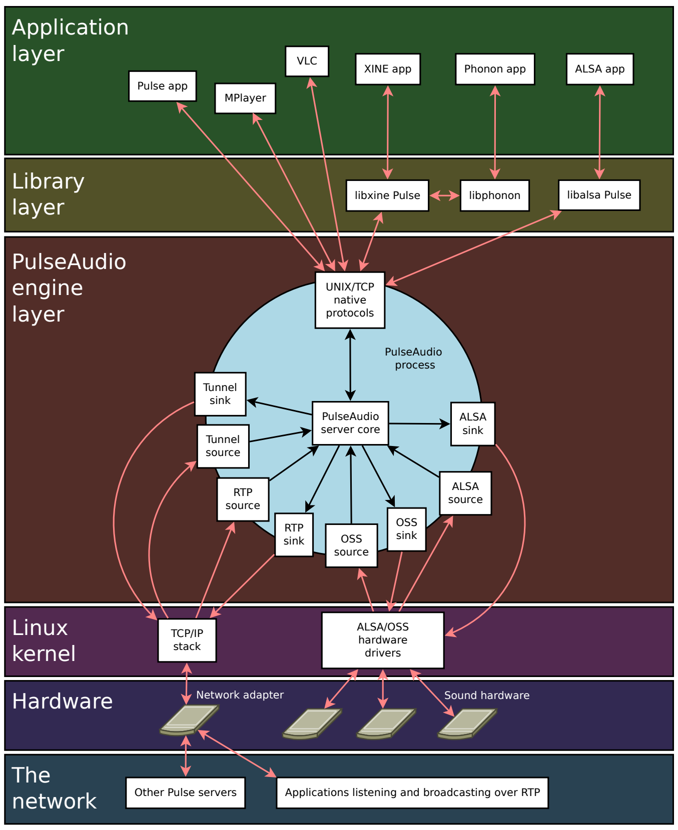

Fig. 2. High-level PulseAudio architecture.

`ProbeManager` interacts with each *Probe* through an object of type `ProbeInterface`, derived from `QxtRPCPeer`, that completely hides the details of the network protocol used to communicate with the remote side, thus greatly simplifying the implementation.

Probe uses DBus to interface with BlueZ [11], [12]. When asked to, it begins monitoring the discoverable devices by calling `BluezAdapter::StartDiscovery()` and then waits to be notified about their presence by the signal `DeviceFound`. While a discovery session is active, the BlueZ daemon also notifies the requesting client each time the detected RSSI (Received Signal Strength Indicator) changes: this allows sending the updated value to the main application as quickly as possible.

B. *PulseAudio module*

The PulseAudio component has three main responsibilities:

- **querying** the daemon for the list of available sinks and audio streams and keeping such information up-to-date;
- **loading** PulseAudio modules with the appropriate configuration and **unloading** them when they are no longer needed;
- **redirecting** an audio stream from one sink to another when requested by the application's logic.

All these operations are performed asynchronously. To keep track of in-flight operations, PulseAudio libraries return a reference-counted `pa_operation` object for each of them.

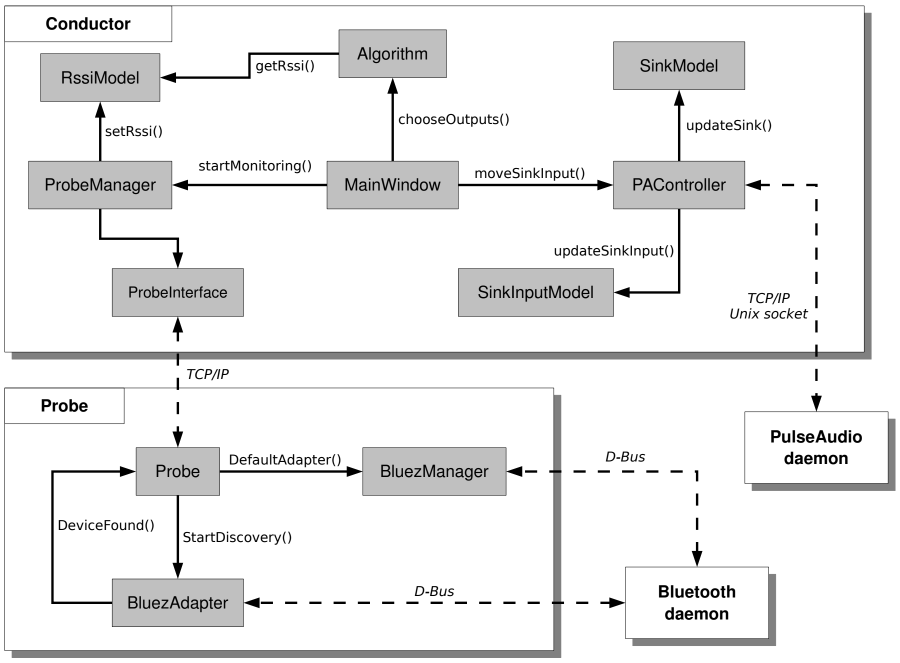

Fig. 3. Main classes and interactions among them.

Although the full introspection API is very powerful, it is also rather complex to use, being written in plain C and heavily relying on callbacks to implement asynchronous notifications. Therefore we developed a thin C++ layer on top of it, to simplify its usage by the rest of the application.

The two basic data structures at the bottom of PulseAudio, `pa_sink` and `pa_sink_input`, are wrapped respectively by the classes `Sink` and `SinkInput`.

Asynchronous operations are wrapped in a similar way: each kind of operation is encapsulated by a different class, which hides all the details about how to submit that operation to the daemon and how to handle the result or a possible error. These classes are then organized into a simple hierarchy, with a common abstract base class (`PAOperation`) that exposes a unified interface to execute operations, without having to worry about their type. The two most interesting `PAOperation` subclasses are:

- `LoadModuleOperation`: loads a PulseAudio module to accomplish a specific task. In particular, our application uses the two following modules:
  - `module-tunnel-sink` creates a new sink that forwards its input to the other end of the tunnel via a TCP connection; it requires a running PulseAudio daemon on the remote side.
  - `module-combine` allows playing the same audio stream through two or more sinks simultaneously. A new virtual sink is allocated and all data written to it is forwarded to every connected sink.
- `MoveOperation`: redirects a `SinkInput` from the current sink to the specified one.

On top of everything, class `PAController` is the façade for the whole module and provides a further level of abstraction. Internally it uses the classes described above, thus avoiding most of the complexity of interfacing with PulseAudio libraries directly. Its API is simple to use and consists mainly of these two methods:

▷ `createTunnel(QByteArray server)`
▷ `moveSinkInput(SinkInput input, QList<QByteArray> speakers)`

The first method executes a `LoadModuleOperation` to create a tunnel sink towards the specified server. The second method redirects an arbitrary `SinkInput` to all the sinks corresponding to the specified list of speakers; this is accomplished using a `MoveOperation`, possibly preceded by the loading of an instance of `module-combine` if the list contains two or more speakers.

In order to implement these features, `PAController` has often the need to perform not just a single atomic operation but many of them. However these operations must usually be executed in a well-defined sequential order. This is quite difficult to achieve directly, due to the asynchronous nature of PulseAudio APIs. Therefore we designed a dedicated class (`PAOperationQueue`) which, while keeping an asynchronous design, is able to safely handle a queue of operations by submitting them only when all their dependencies have been satisfied, i.e. after the daemon notifies that all the operations on which they depend have been completed.

C. *Output selection algorithm*

This algorithm is the core of *Conductor*: it tries to figure out in which room the monitored device is currently located and it selects the speakers that have to start reproducing sound accordingly.

The algorithm runs periodically at configurable intervals and, on each execution, it proceeds as follows. First of all, it obtains the latest RSSI value associated with each room using the method `RssiModel::getRssi()`. Then the values are sorted and a heuristics is applied to the highest ones. This heuristics takes into consideration the relative position of rooms, discarding any invalid combinations from the set of selected outputs (e.g. it does not make any sense to reproduce simultaneously in two rooms which are not contiguous). Finally, if the resulting set of rooms is different from the current one, the `outputsChanged` signal is emitted, eventually causing the invocation of `PAController::moveSinkInput()`.

The following pseudo-code shows more precisely how the implemented algorithm works.

The following list presents a short description of all variables involved in the algorithm:

- `adjacencyMap`: maps each room to the list of its adjoining rooms;
- `best`: a set containing the rooms that have the first `maxSimultaneousSpeakers` highest RSSI values;
- `curRooms`: the rooms where the audio stream is currently being played;
- `maxRetries`: maximum number of retries (see description of `retryCount`); when exceeded the playback should be stopped;
- `maxSimultaneousSpeakers`: maximum number of speakers that can be activated simultaneously;
- `neighbors`: the set of rooms that are contiguous to at least one room in `curRooms`;
- `outputs`: the new set of rooms chosen by the algorithm to reproduce the stream;
- `retryCount`: an integer counter used to delay playback interruption when the device temporarily falls out of range;
- `rssiMap`: maps each room to the last RSSI value received by the *Probe* located inside that room.

Algorithm::chooseOutputs()

```
1: $retryCount \leftarrow 0$
2: $outputs \leftarrow \varnothing$
3: $best \leftarrow \varnothing$
4: $sorted \leftarrow sort(rssiMap)$
5: for $i = 1$ to $maxSimultaneousSpeakers$ do
6:     $best \leftarrow best \cup sorted[i]$
7: end for
8: if $best = \varnothing$ and $retryCount < maxRetries$ then
9:     $retryCount \leftarrow retryCount + 1$
10:    $sleep(updateInterval)$
11:    goto 4
12: end if
13: $neighbors \leftarrow \varnothing$
14: for all $r \in curRooms$ do
15:     $neighbors \leftarrow neighbors \cup adjacencyMap[r]$
16: end for
17: $outputs \leftarrow best \cap neighbors$
18: return $outputs$
```

IV. Prototype testing

A. Hardware requirements

During the project specification and development we assumed the availability of a precise hardware setup.

The prototype basically requires:

- a Bluetooth **device** (e.g. smartphone), used to determine the room in which the person is currently located. For simplicity, we have considered only one such device, but the prototype can trivially be extended to support an arbitrary number of devices;
- a **central server**, where the audio player and the main *Conductor* application run.

Furthermore, inside each room there must be:

- one Bluetooth **adapter**, used to monitor the RSSI of the device. It must support *HCI inquiries with RSSI reporting*, as documented in the Bluetooth HCI specification;
- one **speaker**, that starts reproducing the audio stream when its room is chosen for playback.

B. Software configuration

The user is required to setup some parameters before starting the main application. Currently, we have not implemented any sophisticated configuration methods, in order to keep the prototype simple, but configuration management is entirely encapsulated inside the class `Config`, which makes it trivial to implement an alternative and more flexible backend. For the time being, the values can be directly modified inside `config.cpp` and `config.h`.

Available configuration keys are:

- `maxRetries`: this is directly used in the output selection algorithm to set an upper bound for `retryCount` (see section III-C);
- `maxSimultaneousSpeakers`: maximum number of speakers that can be activated simultaneously;
- `probesAddresses`: maps each room to the IP address of the *Probe* instance that is running there;
- `roomsNames`: list of names of all rooms; this value is automatically obtained from `probesAddresses` and rarely needs to be modified;
- `roomsTopology`: for each room, it gives the set of rooms that are contiguous to it;
- `updateInterval`: the amount of time (in ms) between two consecutive executions of the algorithm.

C. Testing procedure

Due to the lack of appropriate Bluetooth hardware, we were not able to test the prototype as a whole. Nevertheless we did perform some functional tests of the Bluetooth and PulseAudio components separately in a simulated environment.

The testbed consisted of two notebooks equipped with a sound card and a speaker, connected via an Ethernet cable. Each machine represented a different room.

When started in testing mode, the interface of *Conductor* allows to manually set the RSSI values associated with each room. By changing these values, the user can simulate the movements of the monitored device across rooms and check whether the system reacts correctly. Normally the stream should only be reproduced on the machine with a non-empty RSSI. If neither are empty, it should instead be reproduced on both machines simultaneously.

D. Experimental results

Overall the prototype proved to be very stable and efficient during every operation. However we discovered two noteworthy issues, probably to be attributed to the technologies we rely upon and not to our implementation.

The first problem is an inherent characteristic of the inquiry method we employed for device monitoring. The RSSI values reported by one adapter for the same device showed a high variance, which makes it difficult to integrate sophisticated heuristics in the algorithm. This could be a major roadblock for applications that need to determine the precise location of the device, but in our case it was not a serious shortcoming, since we just needed an approximation in order to infer the current room.

The second issue involves the redirection of audio streams. Soon we discovered that PulseAudio's `module-combine` and `module-tunnel-sink` are not mature enough to ensure a flawless listening experience. Indeed sometimes we heard noticeable latencies and even incorrect resampling when an audio stream had just been moved from one sink to another (possibly virtual) one.

V. Future work

A. Real-world testing

The prototype has been tested through an additional user interface that allows to simulate a person's movements. This interface exposes a monitor with an editable input box for each room. By interacting with it, the user is able to insert an arbitrary RSSI value to simulate his movements among the rooms (the higher the value, the closer he is to the corresponding room). However this is just a simulation, thus the overall system behavior might be slightly different in the real-world case.

In order to improve the consistency and the reliability of test results, it is therefore advisable to perform a more thorough testing session in a real-world scenario. We could not do that due to the lack of a sufficient number of Bluetooth adapters satisfying the hardware requirements outlined in section IV-A.

A more realistic testing phase could also have highlighted some shortcomings of the current algorithm and would have enabled us to fine-tune it to meet the initial expectations for the project.

B. Multiple monitored devices

In section IV-A we assumed that there is only one device to monitor. However we believe that extending the system to be able to locate multiple devices would be a major feature. This entails the necessity of evaluating how different devices could interact, and will eventually require the development of a policy to handle conflicts between them. Once a solution to this potential issue has been found, modifying the current implementation should be trivial because the prototype has already been designed with this extension in mind, so the code is already quite generic and makes very few assumptions about the number of monitored devices.

C. Lightweight control application on device

Currently the device is only used to locate the person wearing it. It would be interesting to develop a mobile application to give the user a certain degree of control on audio streams. For instance, with this new feature, he should be able to adjust the volume and pause or stop the playback. A further improvement could be constituted by an interface to navigate through the playlist, i.e. choosing a specific song or skipping to the next/previous track.

VI. Conclusion

The Bluetooth technology has traditionally been considered only as a cable replacement to enable short-range data and voice communications. Our project showed how it can instead be used to detect the position of a person in a contained environment with reasonable approximation. Furthermore, the inquiry method we used resulted in an extremely low power consumption for the monitored device.

PulseAudio proved to be a very advanced sound server, although some of its features were not mature enough to be able to always provide a reliable and glitch-free rerouting of audio streams. In our opinion this is an area of PulseAudio that needs more work.

Overall we believe that our prototype represents a good starting point for future extensions and improvements.

References

[1] G. Marfia, M. Roccetti, "TCP At Last: Reconsidering TCP's Role for Wireless Entertainment Centers at Home", *IEEE Transactions on Consumer Electronics*, Vol. 56, N. 4, Nov. 2010.

[2] G. Marfia, M. Roccetti , "Dealing with Wireless Links in the Era of Bandwidth Demanding Wireless Home Entertainment", in *Proc. of IEEE NIME'10-ICME'10*, Singapore, Jul. 2010.

[3] S. Kim, "Smart Appliances and Services – The Home of the Future Is Here" , in *Proc. of LonWorld 2000*, Orlando, FL, USA, Oct. 2000.

[4] D. Surie, O. Laguionie, T. Pederson, "Wireless Sensor Networking of Everyday Objects in a Smart Home Environment" , in *Proc. of Intelligent Sensors, Sensor Networks and Information Processing 2008*, Sydney, Australia, Dec. 2008

[5] N. Noury, T. Herve, V. Rialle, G. Virone, E. Mercier, G. Morey, A. Moro, T. Porcheron, "Monitoring Behavior in Home Using a Smart Fall Sensor and Position Sensors", in *Proc. of 1st Annual International, Conference On Microtechnologies in Medicine and Biology*, Lyon, France, Oct. 2000.

[6] S. Dagtas, G. Pekhteryev, Z. Sahinoglu, "Multi-Stage Real Time Health Monitoring via ZigBee in Smart Homes", in *Proc. of AINAW'07*, Niagara Falls, Ontario, Canada, May 2007.

[7] D. N. Kalofonos, S. Shakhshir, "Intuisec: A Framework for Intuitive User Interaction with Smart Home Security using Mobile Devices", in *Proc. of IEEE PIMRC 2007*, Athens, Greece, Sep. 2007.

[8] C. E. Palazzi, S. Ferretti, M. Roccetti, G. Pau, M. Gerla, "What's in that Magic Box? The Home Entertainment Center's Special Protocol Potion, Revealed", *IEEE Transactions on Consumer Electronics*, vol. 52, no. 4, Nov. 2006,

[9] BlueZ: the official Linux Bluetooth protocol stack, http://www.bluez.org/

[10] PulseAudio sound server, http://www.pulseaudio.org/

[11] Marcel Holtmann, Playing BlueZ on the D-Bus, www.kernel.org/doc/ols/2006/ols2006v1-pages-421-426.pdf

[12] H. Pennington, A. Carlsson, A. Larsson, D-Bus Specification, http://dbus.freedesktop.org/doc/dbus-specification.html