Integrating multicore awareness functions into distribution middleware for improving performance of distributed audio surveillance

Marisol García-Valls[a]

[a]*Departamento de Comunicaciones, Universitat Politècnica de València, Valencia, Spain*

Abstract

This paper describes an approach to improve the performance of the distributed audio-processing functions for audio surveillance systems. In order to increase portability, current distributed audio-processing uses the default capacities offered by the underlying scheduling facilities of the operating system. In this approach, a set of capacities are added to the distribution software that enable the reduction of the distributed processing time of audio frames at the server side by adding functions that utilize the underlying hardware resources including exclusive core reservation. By loosing some generality in the design of the distribution software, it is possible to increase performance and provide better isolation to selected audio tasks in the presence of other competing software tasks. The approach is designed and implemented as well as analyzed on general purpose computers with a server-client architecture using serial scheduling of the audio tasks and parallelizing the digital signal processing computations. The proposed solution is implemented and analyzed showing benefits in performance and robustness over single threaded audio processing. The resulting system is significantly more robust in the presence of other competing software tasks (noise). These results directly yield the possibility to manage more concurrent audio streams at the server side.

Keywords: software design, timeliness, performance, distribution middleware design, audio processing, multicore awareness

Email address: magarva1@upvnet.upv.es (Marisol García-Valls)

1. Introduction

Surveillance systems are gaining popularity in very different domains such as safety in public and private spaces, eHealth, or industrial control for smart factories; this popularity increase has been enabled by major technological steps such as the transition from analog to digital systems over the Internet Protocol [1] around the beginning of this millenium. Now, modern surveillance systems are progressively moving towards (and becoming part of) the *social dispersed computing* paradigm [2] as there is a need for providing smarter solutions at the cost of boosting their computational complexity. For example, industrial systems are rushing into the Industry 4.0 era in which the software importance raises as it allows industries to provide richer though more complex functionality in fully distributed environments over countless heterogeneous computation nodes. Increased intelligent solutions will allow higher virtual presence of the human-like (intelligent) functions at the factory floor by intensive real-time sensing, monitoring, and control of the industry operation.

Surveillance systems are applications with real-time requirements as they have to detect anomalies and take mitigation actions immediately to avoid problems such as asset destruction or even threat to humans, among other. This implies reacting and handling the failure situation before a specified deadline. The logic run by such systems needs not only the appropriate hardware that provides sufficient computation power such as multicore technology, it also needs to integrate appropriate efficient software layers like operating systems, the distribution software, and even virtualization technology [3] and application-level logic that result in a system with timely execution.

In audio-based processing for video surveillance, pattern matching for a given set of collected audio samples has to be done in real-time to detect anomalous situations. This is illustrated in Figure 1 for a factory floor video surveillance. On the other hand, it should always be considered that real-time audio-processing activities have little tolerance to deadline misses as that yields poor audio quality. Additionally, poor-quality audio will significantly delay the subsequent audio-processing functions and lower-quality anomaly detections will be made. Therefore, in high-quality audio applications, processing deadlines must be meet; having an efficient execution framework that accelerates audio-processing activities is highly desirable for this purpose.

More and more, the new functions need heavy processing of data of different nature, e.g., asset-monitoring data, video-surveillance media, or audio-sampling data, among others. The distribution software layer is key

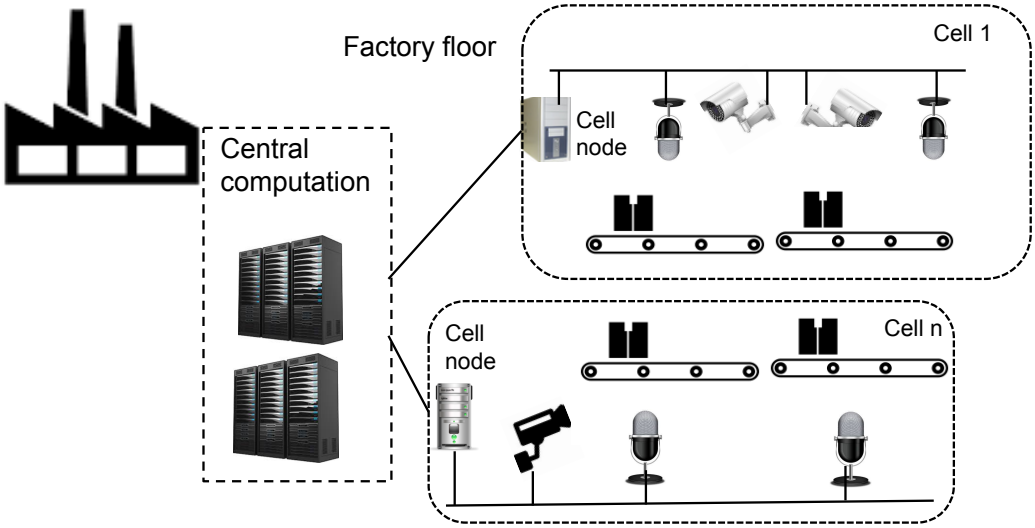

Figure 1: Audio-based surveillance system for failure detection

for realizing these characteristics in a fully distributed and heterogeneous environment, where verification of the system properties [4] will also be needed (especially in critical domains such as cyber-physical systems). Nevertheless, still the design of intermediate distribution layers favors abstraction as opposed to performance; the principles of high abstraction allow highly heterogeneous computation nodes to interact through a common substrate; such a design does not get the most out of the new computation platforms that are based on powerful hardware like *multicore* processors which currently dominate the market. Multicore processors offer the possibility of speeding up the execution of the software, offering dedicated cores to given greedy operations that can be run in parallel.

Depending on the severity level of the timing constraints, the system will have to include either real-time scheduling (i.e., typically for hard real-time and safety-critical domains) or best-effort techniques (i.e., for soft real-time domains). The complexity of the scheduling increases on multicore systems which have more than one processing core, and different approaches have been proposed in order to take advantage of their higher computational capabilities. Most of them rely on assigning tasks in the scheduling queue to a single core based on availability.

A number of solutions for distributed audio transmission exist, but these mostly focus at the signal processing level, being rather silent about the

influence of the underlying communication middleware and the processor characteristics. The effect of the hardware on the timeliness of the application can overcome the (possible) performance drawbacks of the used middleware. To benefit from the computation power of multicore, it is needed also to exploit the parallelization of the audio tasks with a parallelization framework such as OpenMP. Although only partially applied to other domains such as acceleration of eHealth services [5], to the best of our knowledge, there is not a software framework that is highly aware of the hardware characteristics that combines these three elements to improve the efficiency and timeliness of distributed audio-processing: a multicore hardware specifically controlled for the actual processing, a communication middleware for the audio transmission, and a parallelization framework for effectively assigning simultaneous tasks to selected cores.

This paper presents a framework for distributed audio-processing that is highly aware of the underlying multicore processor, improving the timeliness of the processing as compared to the traditional approach, based on the default kernel scheduling and core assignment. Our framework integrates the parallelized signal processing functionality with the distributed communication of the audio samples using off-the-shelf middleware for a low cost (but reliable) solution, and affinities for mapping audio-processing tasks to specific cores. We analyze the performance of a serial task scheduler parallelizing the internal computations of the tasks to reduce their execution time and compare it to the proposed parallelization framework, and we show that it leads to a significant improvement in processing time and robustness in the presence of unrelated system load.

The document is organized as follows. Section 2 describes the baseline technologies for this work. Section 3 presents the architecture and the specific considerations for the distribution facilities. Section 4 describes the design of the application and system for distributed audio processing in the industrial surveillance scenario and describes the interesting details of the implementation. Section 5 describes the evaluation. Section 6 presents a summary of related work. Finally, section 7 concludes by reviewing the improvements of our approach.

2. Parallelization and runtime support

This section provides a brief overview of different technologies for parallelization and scheduling control in Linux.

2.1. Parallel processing infrastructures

OpenMP [6] is a parallelization infrastructure that provides an application programming interface (API) consisting on a set of compiler/preprocessor directives, library routines, and environment variables for shared memory parallel programming [7], which constitutes a language for multi-threaded applications. It uses Pthreads on some operating systems, which favors the development of portable solutions.

Multi-threading may increase significantly the performance of an application with concurrency, as several computations are performed at the same time on different processors. Threads are like processes for the system in terms of execution, they have their own stack and program counter, but they have access to the same virtual memory address space of its parent process. A process can be either single- or multi-threaded. Our system uses threads as they result in more efficient execution given their reduced context switch cost [8]. As threads of a same parent process have a shared memory space, they can communicate through shared memory (global variables); this come at the cost of incurring in potential race conditions. Consequently, the concurrency and parallelization of our framework requires that synchronization constructs are used in order to prevent race conditions and synch errors in general.

An advantage of OpenMP over other parallel programming paradigms is that it can be easily integrated on existing code and some studies suggest that using OpenMP over threads with Pthreads increases the robustness without sacrificing performance [9]. The fork-join model for parallelizing tasks used in OpenMP allows to spawn multiple threads to run a give code block in parallel. In our specific target system, the tasks are parts of the Digital Signal Processing (DSP) that our real-time audio application runs.

OpenMP has three main constructs: compiler directives; run-time library functions; and environment variables. Compiler directives enforce a number of behaviours such as spawning a parallel region; division of blocks of code among threads; distribution of loop iterations among threads; serialization of code blocks; or synchronization of work among threads. Run-time library functions provide a number of useful operations such as setting and querying the number of threads at a given instant; getting a thread's id; querying if the control flow is in a parallel region and at what level; etc. An example of a C/C++ run-time library function is *int omp_get_num_threads(void);*. Environment variables control the execution of parallel code at run-time such as: setting the number of threads (*setenv OMP_NUM_THREADS*

8); indicate how loop iterations are split or divided; enabling and disabling nested parallelism; binding threads to processors; etc.

The most attractive feature of OpenMP for our audio-processing framework is the magic loop parallelization. Element-wise operations on arrays can be performed simultaneously on different processors, and this can be indicated with a compiler directive. The syntax of a compiler directive is `sentinel directive-name [clause, ...]`, such that the *sentinel* varies with the programming language, e.g., for C/C++ it is *#pragma* and for Fortran it is *$OMP*. Code 1 shows the basics of loop parallelization with OpenMP. In the example, all the variables are shared, except the iteration index `i`. The *reduction* directive indicates that each processor maintains a private copy of the shared variable `x`, and that these private copies are combined with the indicated operation at the end of the parallel execution area.

Code 1: Parallelized loop with OpenMP

```
#pragma omp parallel for reduction(+:x)
for (i=0; i < n; i++) {
  c[i] = a[i] + b[i];
  x += a[i];
}
```

The other main infrastructure used for parallelization is MPI [10], which is a language independent specification for interprocess *message passing*. MPI is designed for systems with distributed memory, cluster environments of single-processor machines; and since version 3, it incorporates an extension for shared memory processing. It has several implementations that comply with the specification, like IntelMPI [11] and OpenMPI [12].

Although the current version of MPI is suitable for shared memory applications, OpenMP has a simpler syntax, being easier to use and to debug for shared-memory systems like the current symmetric multiprocessor computers.

Hybrid systems with different memory spaces, with any of them associated with several processors, are a challenge to application design. Existing applications using either OpenMP or MPI may implement the extensions of their current infrastructure to scale to the new scenario. Nevertheless, both platforms are compatible and can be used together carefully to increase the performance [13].

2.2. Execution control and instrumentation

The fundamental functions for controlling the processor are provided by the operating systems and they are: process scheduling and memory management. The default scheduling policy for some operating systems (e.g. Linux) can be customized to suit different applications targets such as real-time. Our framework relies on the kernel supporting functions to control the execution of the audio tasks and to map them to given cores when possible. Table 1 shows the main facilities used for controlling the distributed audio processing.

Table 1: Audio processing support functions

Function	**Purpose**
Scheduling, process management	Prioritize audio processing tasks
Threading and concurrency	Prevent shared audio race conditions
Priorities and processor affinity	Core assignment to audio tasks
Signals	Handle communication interrupts
File I/O	Log of audio tasks operations
Transport level transmission	Audio packets communication
Real-time clocks	Precision delay measurements
Memory management	Efficient consumption avoiding leaks

The proposed audio processing framework uses standard runtime support functions by means of Posix threads (Pthreads) and through OpenMP for parallelizing the audio processing computations. Posix is also used in the core distribution software (Ice) to handle the concurrency when accessing the queue of received packets.

The required high-precision execution-time measurement is performed through `clock_gettime(clockid_t clk_id, struct timespec *tp)` method, selecting a high resolution clock (real time, wall time, or process time) with nanoseconds precision (typically $1ns$ resolution) to obtain the requested timestamp (`timespec`).

The parallelization library and the networking middleware allow us to implement a more efficient and robust application, but these libraries only provide a higher level of abstraction to interact with the operating system. In the end, the operating system is in charge of handling the system calls either of the application or of the libraries to modify the scheduling policies or interact with the hardware.

3. Distributed audio surveillance framework

This section presents the approach for improvement of the timeliness in the audio-surveillance system with a previous description of the core distribution layer that is used to enable the remote interaction across the system nodes.

3.1. Architecture

The proposed system follows a controlled client-server architecture that contains: *cell nodes* which are those in charge of performing the initial audio pattern assessment to detect failures; *central nodes* which perform finer grain and more heavy computations on the audio samples over a much larger audio historic data to detect and anticipate possible failures based on noise patterns; and *audio sensors* which are small embedded computers with microphones to collect data samples (e.g. [14]). This is illustrated in figure 2.

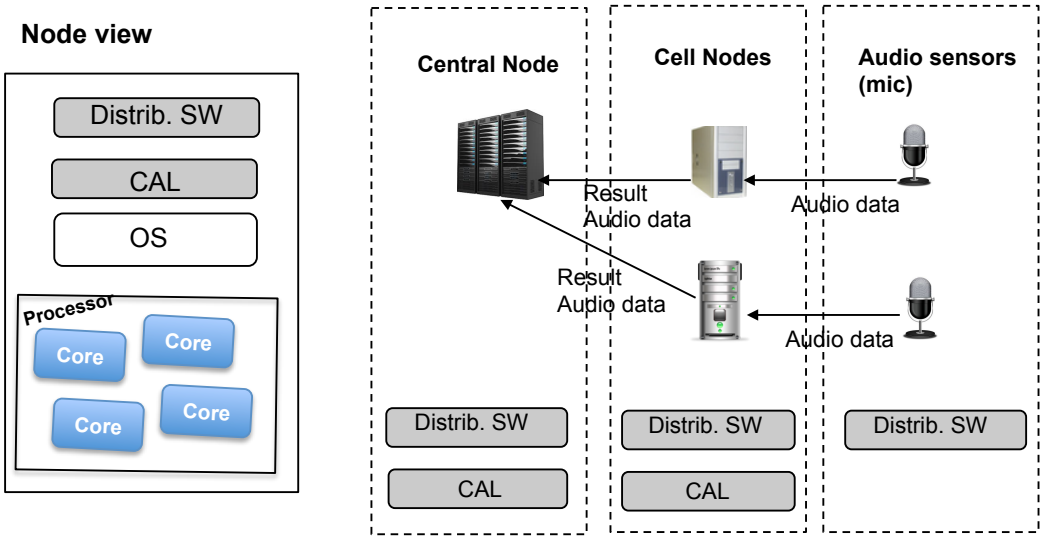

Figure 2: View of the the software layers of a node; Interaction across nodes and software functions.

In this architecture, both the cell nodes and the central node play the role of servers; whereas the audio sensors are clients that collect audio samples and transmit it to the cell node. Cell nodes merge the audio data comming from a number of audio sensors to detect posible failures. It is also the case

that cell nodes act as clients to the central server side when they transmit the collected audio samples.

The software stack of the different nodes differs. As all nodes are distributed, they all run a version of the lightweight distribution software (*Distrib. SW* module on the node view part of figure 2). For those acting as servers (central and cell nodes), they also run the hardware-aware logic (*CAL* module – *Core Assignment Layer*– in figure 2) that provides the parallelization of the audio processing functions to speed up the computations using the available processing cores. This will result in shorter response times in the output generation; outputs may be either the detection of an alarm/anomaly (*alarm*) or a sucess operation (*ok*).

Figue 3 shows the software layers that are present at the nodes (both at the server and client sides) of the audio-surveillance deployed nodes.

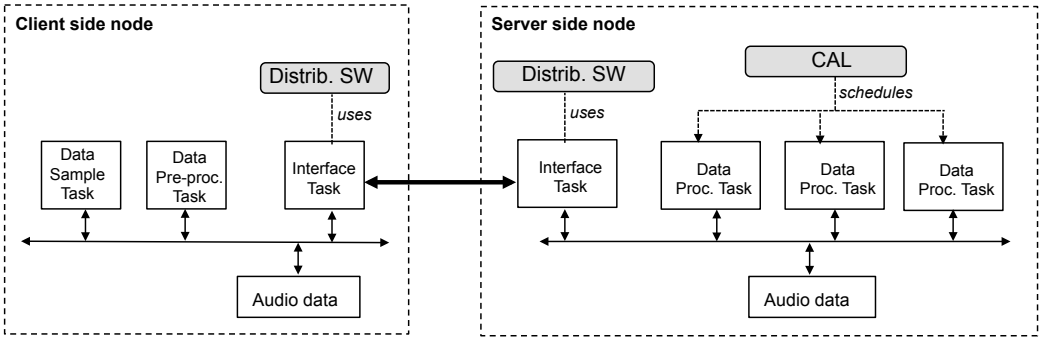

Figure 3: Task interaction through the framework modules.

The distribution layer is present for all node types; the hardware assignment module is only present at the processing side of nodes having a server role.

CAL module uses the standard POSIX runtime support functions of the underlying operating system. The priorization of tasks and the assignment of tasks to specific cores use the processor bitmasks through affinities. CAL hooks to OpenMP for parallelizing the audio processing computations. CAL arbitrates the assignment of the audio processing tasks, enforcing their allocation to specific cores. Where CAL is present, it provides hardware core reservations.

The *Distrib. SW* module is available at all the participating nodes as they are part of a fully distributed system where all nodes need to exchange data. This distribution module handles the server incoming concurrency by restricted access to received-packets queue. It provides the communication facilities through remote invocations through the Internet Communications

Engine (Ice), used as the core distribution software. The design details of both modules, CAL for hardware awareness and *Distrib. SW* based on Ice, are presented in the next sections. The communication across nodes is performed by specific interface tasks that use the *Distrib. SW* module.

3.2. Core distribution software

The distribution module uses Ice [15] as core distribution software because of its lightweight structure that uses standard operating system facilities and has operating system standard compliance (e.g. it has a POSIX compliant runtime). Its lightweight design and performance has been evaluated extensively in different works such as [16] and its adaptation facility has been shown in [17], among others.

Each node of a system can take the role of client and/or server. The communication among nodes is agreed through the specification of remote interfaces in an IDL (Interface Definition Language) named Slice, that is independent from the programming languages. This is done through *modules* that are the means to control name spaces and *interfaces* to define the server functions that are available for clients. Code 2 shows an interface definition (of name *AudioProcessing*) that, in this case, receives an input parameter of type `audiostream` which is a sequence of bytes that are indeed the collected audio samples transmitted by the audio sensors. The interface also shows the posibility of requesting a priority audio processing activity through the *prio* parameter. Priority requests will be handled accordingly by the server, through core reservation.

Code 2: Interface definition for a data process function

```
module ServerOp {
 sequence<byte> AudioData;
 interface AudioProcessing {
   void audio_process(AudioData audiostream, int prio);
 };
};
```

Establishing the communication between nodes requires the definition of a *communicator* object; this object is the entrance point to the core distribution software that enables access to communication resources, including the available thread pool to handle the connections asynchronously. The `communicator` allows to create *adapter objects* that handle the interfaces of

the remote objects at the server. An adapter is bound to a port where it listens for incoming requests. Finally, a *servant object* of the interface needs to be instantiated to process the requests handled by the adapter. Figure 4 illustrates the design of the audio processing interfacing through the core distribution software.

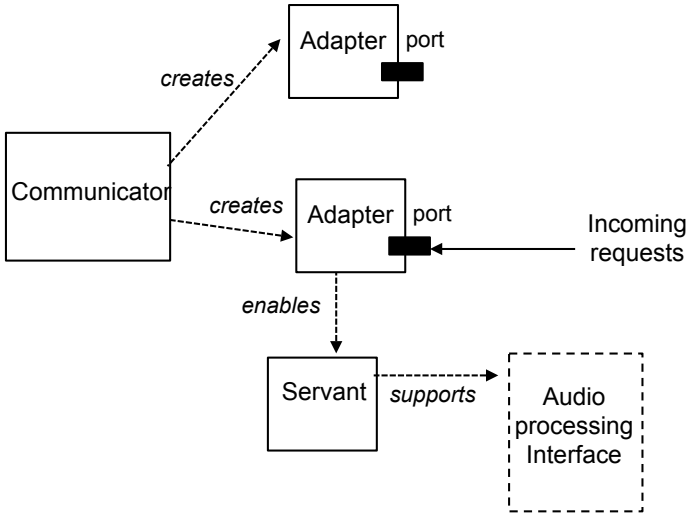

Figure 4: Design of the audio interfacing scheme.

The distributed services at the server side are two functionalities that are remotely accesible: the *audio service* for processing the streams received from the clients; and a *noise service* for noise generation upon request of a client. The service that generates noise is used to introduce varying load conditions on server machines; it is an internal hook for instrumenting different possible conditions at the server side (that can later map to different validation conditions) in order to derive the degree of robustness of the system; this can be done by varying the noise level and processing times without affecting the server. The servant supports the execution of the operations listed in the interface provided in Code 2 that contains *audio_process* function. Also, this servant supports the execution of an additional interface named *ServerHook* which contains an additional function *noise_condition_generation*.

Resource efficiency is achieved in different axes. On the one hand, the distribution module performs memory management collection by reference counting to object instances. Objects with no remaining active references are garbage collected. On the other hand, the *resource allocation in initialization*

pattern is used for efficiency reasons, e.g., deadlocks due to unlocked mutexes can be easily avoided with a `Lock` class that receives the mutex variable on the constructor and automatically acquires the lock; by the same token, it automatically releases the lock on destruction when the scope of the variable terminates.

4. Hardware-aware execution enforcement

4.1. Audio data considerations

The digital audio samples are sent over the network in data packets. The continuous channel of analog systems can be implemented by sending samples at the appropriate high rate. It should also be considered that the design of data networks introduces a fixed size overhead to every packet due to the headers of the protocols at the different network layers. Another important consideration is the damage that can be derived from the loss of a packet with a number of audio samples. In this situation, there is a tradeoff between efficiency and robustness, and the different coding schemes choose various configurations. For instance, the Advanced Audio Coding (AAC) standard uses 1024 frames per packet. A frame has an audio sample per audio channel; in a stereo configuration, it has 2 samples.

The proposed system will use the same number of frames per packet than AAC for a Pulse-Code Modulation (PCM) stereo signal of 16 bits depth and a sampling rate of 44.1 kHz. The communication will be unidirectional, where several clients can send their audio data to the server. The server will handle the concurrent connections, and it will process the scheduling of the audio samples efficiently to guarantee a bounded delay on the audio streams. The audio streams connected to the system are the tasks, and each packet of frames received releases a job of the sender stream task.

4.2. Distributed setting

All participating nodes use the *gflags* library [18] for activation. This results in a flexible design that allows the nodes execution environment to be easily parametized at runtime. The specific entry ports (as shown previously in figure 4) as well as the core enforcement for the audio tasks and their parallelization is set at runtime. The server side can then specify:

- *The listening port* with the `--port` flag.

- *Parallelize the DSP* using the `--openmp` flag. The default is single threaded DSP.

Similarly, the client uses the following flags:

- *The server host and port* with the `--host` and `--port` flag, respectively. The default server is *localhost*.
- *The audio sampling frequency* of the audio stream, that defaults to 44100 Hz, can be specified with the `--freq` flag.
- The `--noise` flag to *generate noise in the server* at different levels for a number of seconds at each level (the default value is 10 seconds).

The distributed processing of the audio streams is deployed as follows. The audio processing service is implemented by the server application, and there is a client side that generates the audio samples sent to the server. The client sends CD quality stereo stream generating vectors of 1024 frames every 0.0232 seconds, that maps to a rate of 44100 frames per second. A frame contains a 16 bit PCM sample of each audio channel, codified for the stereo system as a 32 bit integer.

4.3. Distributed audio processing services

Audio is collected by clients that periodically send the sampled audio packets as a byte stream of audio frames and a unique identifier of the stream. As different audio streams can be sent to a server, an identifier of client and stream is used to differentiate streams. Also, different streams from the same client can be supported in this way. This is easily enhanced to obtain a universally unique identifier by adding the machine IP address. The server handles the requests asynchronously. The `processSamples(int client_id, std::vector<int32_t> audio_frames)` method creates a tuple with the client identifier and the vector of audio frames and appends it on a FIFO (First-In First-Out) queue.

The audio-processing server node runs as a daemon application listening at a port where the clients send the audio samples. In order to provide real-time audio processing, the server exploits the resource management functions of the kernel to control the execution on the multicore processor.

The `AudioServiceI` is the interface created for the server that extends the `IceUtil::Thread` class (the class that Ice libraries define that is a wrapper

to the type POSIX thread). Hence, upon the creation of an instance of this class on the server, the thread has to be started. The class uses an `IceUtil::Monitor` variable (the class that Ice libraries define that is a wrapper to a POSIX condition variable) that makes the thread sleep while the queue of pending packets to be processed is empty. Upon enqueuing a new tuple, the queue monitor is signaled to wake up the thread. Race conditions on the access to the queue are avoided with a mutex that is acquired before accessing to ensure mutual exclusion is preserved.

The thread of the audio processing is scheduled with priority; it is, in fact, the thread that executes the servant code. After it starts execution, the `AudioServiceI` function starts processing audio samples. Audio processing is performed in a loop over the queued audio packets until it is stopped. Stopping is implemented by an additional function `stop()` of the same interface. Initially (at the first execution of the loop), the queue is expected to be empty. Then, the thread will sleep until the first packet of audio frames arrives that will automatically wake the thread up. To avoid abrupt termination after calling `stop()`, the thread loops to process the already queued packets and then joins the server main thread. Similarly, in order to avoid enqueuing packets when the thread is not active, method `processSamples` checks an internal variable that controls the frame processing loop before a new packet is queued.

The DSP of the audio frames runs a nested loop. The outer loop updates a control variable that executes the part of the DSP that is not parallelizable. The inner loop performs element-wise operations on the frames vector as these are parallelizable regions. This inner loop is parallelized with a OpenMP *pragma* directive like the one showed in Code 1.

Code 3 partially shows the logic to perform core assignment for evaluating the robustness in the presence of different load conditions. The pool of threads that are created as per the internal server hook are assigned core affinities to measure the actual execution times and interference for the audio processing tasks. Different load conditions due to external activities running in the processor are simulated with a noise control function that runs a set of synthetic activities for the sole purpose of generating interference as would happen in a real deployment. These different load conditions are controlled through explicit activation and deactivation and are generated in given cores as sketched in Code 3.

Code 3: Robustness validation through controlled noise generation and corresponding core assignment

```
noiseLevel = loadLevel;
for (int i = 0; i < sysconf(_SC_NPROCESSORS_ONLN); ++i)
    {
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(i, &cpuset);
  NoiseThreadPtr t = new NoiseThread;
  pthread_t t_id = t->start().id();
  pthread_setaffinity_np(t_id, sizeof(cpuset), &cpuset)
      ;
}
```

Different load conditions at the server are created by designing an internal hook that can dynamically activate a service *DummyService* to validate different execution conditions that affect the processor load; this is done by creating a pool of threads with a `doNoise()` method inside the former service. One thread per core is created and started in the system and the core affinity binding is set for all the threads to different processors. Threads perform operations that merely consume processor cycles with a microsecond idle time in between consecutive repetitions of the same set of operations. The complexity of the generated load is controlled with a parameter to set the noise level; this parameter can be modified by any client with a call to `doNoise(int noise_level)`. When the maximum specified noise level is set the voluntary waiting of the noise threads is skipped and they consume all its time slice on the processor before being preempted. Similarly, if the noise level is set to zero, the threads exit the noise generation loop and terminate their execution.

5. Evaluation

This section presents the performance evaluation and analysis of the proposed framework for distributed audio processing that is highly aware of the underlying hardware based on multi-core and the usage of parallelization software for execution of selected operations in specific cores.

The execution environment for the multi-core audio processing is based on a computer equipped with an Intel i7-5600U processor at 2.60 GHz with 2 cores and hyperthreading, meaning that 4 threads can run in parallel. The operating system is the 64 bits version of Debian 8 "Jessie" with the version

3.16 of the Linux kernel. In this paper, we focus on evaluating the processing time.

Our validation scenario uses the same number of frames per packet than AAC for a Pulse-Code Modulation stereo signal of 16 bits depth and a sampling rate of 44.1 kHz. The communication will be unidirectional, several clients can send its audio to the server side that will have to handle the concurrency of the connections and process the scheduling of the audio samples efficiently to guarantee a maximum delay on the audio streams. The audio streams connected to the system are realized with active tasks and each packet of frames received releases a job of the sender stream task. The delay of audio processing introduced by the network or by the queuing of incoming packets on the server is not considered. The network delay is mostly related to factors not directly involving the system design; therefore, we analyze the actual performance of the server software stack.

The parallelization can be configured either dividing the computations statically or dynamically, and with different number of threads. The most straightforward solution is to split the computations in equal parts with one thread per CPU in order to maximize the utilization of the system and load balancing. We have found experimentally that there is no significant difference between static or dynamic splitting and the best performance is achieved with one thread per CPU. Although this is true for the best case, we found that dynamic splitting leaving one core free is a more robust configuration that leads to a better worst-case processing time for all noise levels with a minimum difference in the best case performance.

Figure 5 shows the boxplots for different OpenMP configurations for parallelization.

In figure 5, for each noise level in the x axis, each configuration expressed in the legend is provided: the left most box corresponds to the *static* configuration; the middle box presents the *dynamic (all processors)* configuration; and the right most box provides the *dynamic (1 processor)* configuration. The distribution of the processing times with dynamic splitting and leaving one core free is significantly narrower and hence, more robust to noisy environments. The introduced noise is realized by activation of unrelated tasks competing with the audio tasks for the hardware resources as consumption of processor cycles.

A file log is used for storing the temporal time taken by the server to process audio packets within the stream ID and the noise level (i.e., the actual load) at the server. Precision measurements are obtained by collecting timestamps before and after frame processing using the POSIX clock API

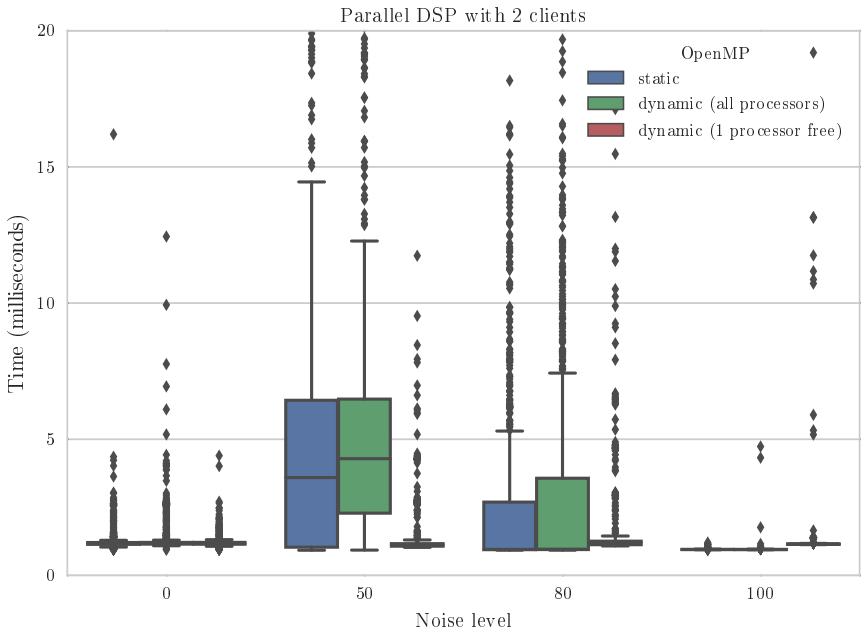

Figure 5: Packet processing time (in milliseconds) for different DSP configurations with two concurrent audio streams. Noise level is expressed in percentage.

(`clock_gettime()` method, described in section 2.2), using `CLOCK_MONOTONIC`. Performance results and analysis is shown in the following section for the collected processing costs.

The capacity of the system (i.e., the maximum number of streams that the server can process simultaneously) is now analyzed from the processing-time measurements. Using a notation compatible with that of real-time systems, each stream is a task and its packets are the jobs. Therefore, the system is schedulable if all the tasks can be completed within their deadline, that may coincide with their period. This depends on the quality of the streams and the actual audio processing of the streams. In a system where all the streams have the same audio quality and the DSP to perform on them is the same (i.e., meaning their processing times and period of the streams are the same), the capacity is given by the equation that is given as follows.

$$N \leq \frac{1024}{f_{audio} \cdot t_{packet}} \tag{1}$$

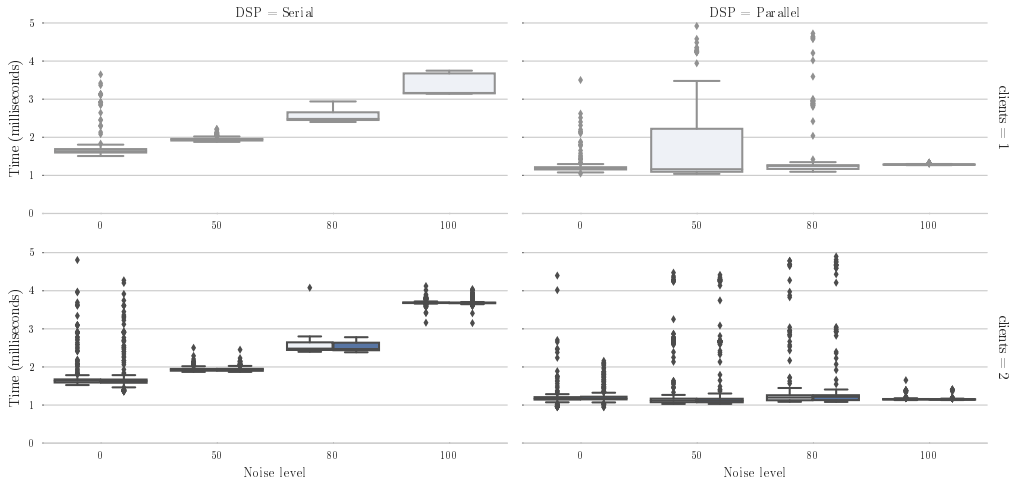

Figure 6: Comparison of the packet processing time (in milliseconds) with serial and parallelized DSP with one and two concurrent audio streams. Noise level corresponds to overall processor load and is expressed in percentage.

where N is the capacity in streams of the system, f_{audio} is the audio stream frequency (44100 Hz by default), and t_{packet} is the time taken by the server to process a packet of audio frames. A stream generates an audio frame every $1/f_{audio}$ seconds and are sent to the server in groups of 1024 frames per packet.

Experimentally, as the processing time is not deterministic on general purpose systems, we can use the $(100-\delta)$ percentile, where δ is the tolerance of the system to packet losses. For stereo audio of 44100 Hz and 16 bit PCM/channel with the DSP of the test application and a tolerance to losses of 25%, we estimate a capacity of 6 streams for the serial DSP and up to 17 when parallelizing the DSP, depending of the parallelization scheme. We use the higher value of the 75-percentile among the different noise levels.

Figure 6 compares the serial and parallel DSP performance for different noise levels, numbers are provided in table 2. We measure the processing times for one and two concurrent streams showing that there is no difference in the processing times between concurrent streams due to the serial scheduling. The figures show that parallel processing, with the proper configuration, is more robust to the noise produced by unrelated system load. The experiments report true values to expose the real effect over thousands of the executions (precisely, 4000 runs). Under very low load conditions and absence of noise,

Table 2: Processing times of the audio packets in milliseconds

Noise level	0		50		100	
DSP	Serial	Parallel	Serial	Parallel	Serial	Parallel
mean	1.77	1.22	1.94	1.56	3.60	1.18
std	0.47	0.24	0.05	1.27	0.21	0.97
min	1.36	0.94	1.87	1.03	3.14	1.13
25-perc.	1.59	1.15	1.91	1.08	3.67	1.15
median	1.64	1.17	1.94	1.15	3.68	1.16
75-perc.	1.67	1.21	1.96	1.18	3.69	1.28
max	5.43	4.40	2.50	19.99	4.12	19.20
Number of measures	925	1150	1262	1207	1268	1269

the kernel runs all active tasks in the same core as it reduces data dependency effects. This results in situations with higher context switch effects and the overall time is typically larger than when execution is bound to given cores. When tasks start to be confined to selected cores, interference decreases and the overall processing times improve. This shows the set of outliers in both cases, for the serial and the parallel case. The important aspect here is to show the robusness of the approach. In this case, the 75 percentile of average times is low (1.18ms for 50% interference and 1.28ms for 100%) which shows the real benefit of the approach. As we show real numbers, the maximum times are also shown (19.99 and 19.20ms, respectively) but the statistical study of the system behavior shows that they are simply a few outliers out of the thousands of measurements that are reported for each experiment.

The presence of outliers is considerably small as compared to the general execution results. Alghough this would not be suitable for hard real-time systems (in which every deadline must be fulfilled), it is suitable for this type of systems that are soft real-time: it is possible to obtain a statistical analysis that concludes supports the claim of increased robustness. We explain this increase in robustness with the fact that the engineered system is based on a parallelizing infrastructure that helps to reduce the waiting time of tasks, i.e., the time that tasks are waiting to be dispatched into a core. Even the parallel threads altogether suffer a total number of preemptions equal to those of a single process performing the same task. The waiting

times caused by the preemptions can be dramatically reduced with a flexible parallelization configuration as the one provided by our proposed system.

6. Related work

Highly efficient audio processing frameworks have mostly targeted on improvements to the digital signal processing aspect as evidenced by a recent survey on audio surveillance [19]; this is the case of [20] where improvements on mix channel signal separation for audio improvement are presented; [21] that provides a system for localization of people through speech signal recognition based on microphone arrays; or [22] for detection of human audio signals in noisy environments. Another important path for audio surveillance contributions focused on its combination with images to track people in mobile deployments such as [23] for passenger elevators surveillance, or [24] for audio and video surveillance over mobile communications. Consequently, it can be said that the design and development of audio surveillance systems has not focused sufficiently on the improvement of the software stack and its efficient execution over the general purpose computer platforms. The audio processing frameworks that have in deed considered the software stack are mostly related to our proposed framework. However, these have typically been silent about response time improvements, and have not sufficiently considered the mechanisms for controlling the execution over the hardware resources.

It should be noted that some audio processing such as [25] have provided platform-independent libraries for facilitating the implementation of multi-threaded real-time audio applications for multiple input/output audio channels. In such a contribution, a simple though static number of threads for audio processing, supporting both real-time and non real-time requirements is provided. Other Java-based contributions such as [26] and [27] have been directly based on the contributed Linux audio drivers. Other media-centric approaches are also available in the literature such as [28] that offers a framework for functional composition of different media streams.

To the best of our knowledge, there are no contributions that aim at integrating the distribution software side with the logic to achieve control over the assignment of the processing resources of the available multicore processors. Although there are some very efficient distribution software designs such as [29] (that supports real-time video transmission over dynamic distributed service-oriented systems) and has been applied in video-surveillance

[30] as well as eHealth [31], etc., it is unaware of the underlying structure of nowadays multicore processors.

7. Conclusions

In this paper, we have presented the design, implementation, and evaluation of a distributed audio-processing system for surveillance.The design of the distribution software is provided with a client-server architecture that is highly aware of the underlying multicore nature of the processor. The system performs a serial scheduling of the audio tasks, parallelizing the digital signal processing computations by integrating a parallelization infrastructure like OpenMP with distribution middleware for the remote transmission of the audio packets across the floor shop nodes and to the central server nodes. The obtained performance evaluation of the proposed system integrates the temporal cost of the audio processing and the actual software stack (i.e., the operating system and the distribution software). The obtained performance is compared with the traditional approach of single threaded audio processing on monocore. Results show that our framework (which integrates the parallelized signal processing functionality with the distributed communication of the audio samples and affinities for audio-processing tasks over specific cores) leads to a significant improvement in processing time and robustness to unrelated system load.

Parallelizing with a dynamic threading scheme and leaving some of the system resources available for unrelated tasks results in the best configuration to improve the worst case processing time. We explain these results by the fact that this configuration better reduces the waiting time of threads, i.e., the time that threads are idle waiting for the dispatcher to assign them a core.

With the obtained results, we prove the processor intensive task of distributed audio-based surveillance, and digital signal processing in general, can benefit from the speed up effect of modern multicore CPU architectures. The improvements translate into a significant increase of the system capacity.

Acknowledgement

This work has been partly funded by the Spanish Ministry of Economy and Competitiveness under grant TIN2017-86520-C3-2-R (*Sistemas Informáticos Predecibles y Confiables para la Industria 4.0*). I wish to acknowledge the

work of Antonio Pastor in the programming of the experiments and data gathering.

References

[1] C. Peng. (2003) Introduction to video surveillance systems over the internet protocol. Texas Instruments. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.122.1149&rep=rep1&type=pdf

[2] M. García-Valls, A. Dubey, and V. Botti, "Introducing the new paradigm of Social Dispersed Computing: Applications, technologies and challenges," Journal of Systems Architecture, 2018.

[3] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," Journal of Systems Architecture - Embedded Systems Design, vol. 60, no. 9, pp. 726–740, 2014.

[4] M. García-Valls, D. Perez-Palacin, and R. Mirandola, "Pragmatic cyber physical systems design based on parametric models," Journal of Systems and Software, vol. 144, pp. 559–572, 2018.

[5] M. García-Valls, C. Calva-Urrego, and A. García-Fornes, "Accelerating smart eHealth services execution at the fog computing infrastructure," Future Generation Computer Systems, 2018.

[6] The OpenMP® API specification for parallel programming. Accessed 2017. [Online]. Available: http://www.openmp.org/

[7] B. Chapman, G. Jost, and R. Van Der Pas, Using OpenMP: portable shared memory parallel programming. MIT press, 2008, vol. 10.

[8] R. Love, Linux kernel development. Pearson Education, 2010.

[9] B. Kuhn, P. Petersen, and E. O'Toole, "OpenMP versus Threading in C/C++," Concurrency: Pract. Exper, vol. 12, pp. 1165–1176, 2000.

[10] MPI Forum. (2015, June) MPI: A Message Passing Interface, version 3.1. [Online]. Available: http://www.mpi-forum.org/

[11] MPI Intel, "Benchmarks: Users guide and methodology description," Intel GmbH, Germany, vol. 452, 2004.

[12] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004, pp. 97–104.

[13] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on. IEEE, 2009, pp. 427–436.

[14] M. García-Valls, J. Ampuero-Calleja, and L. L. Ferreira, "Integration of Data Distribution Service and Raspberry Pi," in Green, Pervasive, and Cloud Computing: 12th International Conference, GPC 2017, Cetara, Italy, May 11-14, 2017, Proceedings. Springer International Publishing, 2017, pp. 490–504.

[15] M. Henning and M. Spruiell, Distributed programming with Ice. ZeroC, 2003, vol. 3.

[16] M. García-Valls, C. Calva-Urrego, J. A. de la Puente, and A. Alonso, "Adjusting middleware knobs to assess scalability limits of distributed cyber-physical systems," Computer Standards & Interfaces, vol. 51, pp. 95–103, 2017.

[17] M. García-Valls and C. Calva-Urrego, "Improving service time with a multicore aware middleware," in Proceedings of the 32^{nd} ACM/SIGAPP Symposium on Applied Computing (SAC), Marrakech, Morocco, April 2017.

[18] The gflags library for commandline flags. [Online]. Available: http://gflags.github.io/gflags/

[19] M. Crocco, M. Cristani, A. Trucco, and V. Murino, "Audio surveillance: a systematic review," CoRR, vol. 1409.7787, 2014.

[20] A. Ozerov and C. Fevotte, "Multichannel nonnegative matrix factorization in convolutive mixtures for audio source separation," IEEE Transactions on Audio, Speech, and Language Processing, vol. 18, no. 3, pp. 550–563, March 2010.

[21] N. D. Gaubitch, W. B. Kleijn, and R. Heusdens, "Auto-localization in ad-hoc microphone arrays," in 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, May 2013, pp. 106–110.

[22] A. Loytynoja and P. Pertila, "A real-time talker localization implementation using multi-phat and particle filter," in 2009 17th European Signal Processing Conference, August 2009, pp. 1418–1422.

[23] T. W. Chua, K. Leman, and F. Gao, "Hierarchical audio-visual surveillance for passenger elevators," in MultiMedia Modeling: 20th Anniversary International Conference, MMM 2014, Dublin, Ireland, January 6-10, 2014, Proceedings, Part II, C. Gurrin, F. Hopfgartner, W. Hurst, H. Johansen, H. Lee, and N. O'Connor, Eds. Cham: Springer International Publishing, 2014, pp. 44–55.

[24] H. Zhao, K. Yin, and Y. Wu, "Design and implement of variable rate audio and video surveillance system based on DM642 in mobile communication network," in 2010 International Conference on Internet Technology and Applications, August 2010, pp. 1–3.

[25] M. Geier, T. Hohn, and S. Spors, "An open-source C++ framework for multithreaded realtime multichannel audio applications," in Proceedings of the Linux Audio Conference, vol. 2012, 2012.

[26] K. van den Doel and D. K. Pai, "Jass: a Java audio synthesis system for programmers," in Proceedings of the 2001 International Conference on Auditory Display, Espoo, Finland, 2001.

[27] N. Juillerat, S. M. Arisona, and S. Schubiger-Banz, "Real-time, low latency audio processing in java," in Proceedings of the International Computer Music Conference, Copenhagen, Denmark, 2007.

[28] S. Müller, S. Schubiger-Banz, and M. Specht, "A real-time multimedia composition layer," in Proceedings of the 1st ACM workshop on Audio and music computing multimedia. ACM, 2006, pp. 97–106.

[29] M. García-Valls, I. R. Lopez, and L. Fernández-Villar, "iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems," IEEE Trans. Industrial Informatics, vol. 9, no. 1, pp. 228–236, 2013. [Online]. Available: http://dx.doi.org/10.1109/TII.2012.2198662

[30] M. García-Valls, P. Basanta-Val, and I. Estévez-Ayres, "Adaptive real-time video transmission over DDS," in 2010 8th IEEE International Conference on Industrial Informatics, July 2010, pp. 130–135.

[31] M. García-Valls and I. E. Touahria, "On line service composition in the Integrated Clinical Environment for eHealth and medical systems," Sensors Journal, 2017.