Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica
Master of Science in Engineering of Computing Systems

CoAP-based Real-time Transport and Synchronization of Sensor Data in Android

Relatore: Ing. Matteo CESANA

Tesi di Laurea di:
Marco ZAVATTA Matr. 766437

Anno Accademico 2012-2013

Alla mia famiglia

Acknowledgments

The main part of this work has been conducted during a six-month internship at the Networks, Security and Multimedia department of TELECOM Bretagne [https://www.telecom-bretagne.eu/research/departments/networks-security-multimedia/], funded by the research project Zewall [http://zewall.eu], under the supervision of Ahmed Bouabdallah, Houssein Wehbe and Bruno Stevant. I want to thank them, as well as Prof. Cesana, for their trust and essential support.

Abstract

In recent years we have witnessed a dramatic growth in the availability of mobile devices with wireless communication and sensing capabilities, such as wireless sensor nodes and smartphones. These are often resource constrained, above all in terms of power supply. Innovative usages can be envisioned, from medical monitoring to entertainment, if these devices become seamlessly interconnected in a so-called Web of Things. Interacting via sensors to the physical world means that applications often have strict real-time and delay requirements. In the context of smartphones, low-delay communication is made possible by latest cellular technologies. This work focuses on developing a live streaming service of sensor data from a mobile source to a generic client on the Web, supporting also synchronization to other media types. It aims to serve as a proof of concept on the suitability for this class of services of the Constrained Application Protocol (CoAP), a new protocol under standardization designed to be a key enabler of the Web of Things.

Sommario

Negli ultimi anni si è assistito ad una rapida crescita nella disponibilità di dispositivi mobili in grado di comunicare in modalità wireless ed equipaggiati con numerosi sensori, ad esempio i nodi sensore e gli smartphones. Questi normalmente dispongono di risorse limitate, specialmente in termini di consumo di potenza. Molte applicazioni innovative possono essere concepite, anche in campi molto diversi tra loro come la telemedicina e l'intrattenimento, se questi dispositivi vengono interconnessi nel cosiddetto Web degli Oggetti (Web of Things). L'interazione con l'ambiente per mezzo di sensori porta queste applicazioni ad avere in molti casi dei requisiti di ritardo e di esecuzione in tempo reale. Nel contesto degli smartphones, la comunicazione a basso ritardo è resa possibile dalle moderne reti di accesso cellulare. Questo lavoro è incentrato sullo sviluppo di un servizio di diffusione in diretta verso il Web di misurazioni effettuate dai sensori di un dispositivo mobile, supportando la sincronizzazione del flusso con altri generici formati multimediali. Si pone l'obiettivo di dimostrare la validità per questa tipologia di servizio del Constrained Application Protocol (CoAP), un protocollo in fase di standardizzazione che mira ad essere un fattore abilitante allo sviluppo del Web of Things.

Contents

Abstract II
Sommario III
List of Figures VI
List of Tables VIII

1 Introduction **1**

2 Synchronization and Real-time Transport **4**

2.1 Multimedia Synchronization 4
2.1.1 Streams and Sampling Frequency 7
2.1.2 Synchronization Metrics 8
2.1.3 Poor-man Synchronization 9
2.1.4 Metadata-based Synchronization 10
2.2 Real-time Transport over a Network 14
2.2.1 Delivery Deadline 15
2.2.2 Elastic Buffer 16
2.2.3 UDP Transport 17
2.2.4 Error Correction 18

3 Application Protocols **21**

3.1 Real-time Transport Protocol (RTP) 21
3.2 Constrained Application Protocol (CoAP) 24
3.3 Message Queue Telemetry Transport Protocol (MQTT) 27

4 Design Choices and Implementation **30**

4.1 Measuring Quality of Experience . . . 30
4.2 Synchronization Algorithm . . . 31
4.3 CoAP as Application Protocol . . . 36
4.3.1 Introducing Synchronization Semantics in CoAP . . . 38
4.4 System Architecture . . . 46
4.5 Server Implementation . . . 50
4.5.1 Sensors in Android . . . 51
4.5.2 Timestamp Assignment . . . 61
4.5.3 Streaming Manager . . . 61
4.5.4 CoAP Manager . . . 66
4.5.5 Message Buffers . . . 68
4.6 Client Implementation . . . 70

5 Experimental Evaluation **72**

5.1 Experiment Variables . . . 72
5.2 Parameter Setting . . . 74
5.3 Results . . . 75

6 Conclusions **83**

Bibliography **86**

List of Figures

2.1 Live Multimedia System 4
2.2 Intra-stream Synchronization 6
2.3 Skew Accumulation 9
2.4 Late Information Unit Arrival 10
2.5 The Wall Clock 12
2.6 Inter-stream Synchronization 13
2.7 The Timestamp Zone 14
2.8 Elastic Buffer Principle 17
2.9 Forward Error Correction by Repetition 19
2.10 Error Correction Window of Opportunity 20

3.1 RTP Header Format 22
3.2 IETF Multimedia Protocol Stack 23
3.3 CoAP Exchange 25
3.4 CoAP Message Format 25
3.5 CoAP Discovery Example 27
3.6 Thin-server Architecture 28
3.7 MQTT Publish Message Format 29

4.1 Clock Mapping 32
4.2 Synchronization Example 35
4.3 Alterations to Stream Time 36
4.4 CoAP Extension 44

4.5 CoAP Extension Payload Format 45
4.6 System Architecture 46
4.7 CoAP Exchange in this Work 48
4.8 System Software Architecture 49
4.9 Server Architecture 50
4.10 The Android Sensor Stack 52
4.11 Generic Sensor Sampling Model 55
4.12 Sampling Random Variability 56
4.13 Simplified Sensor Sampling Model 56
4.14 Sampling Results 58
4.15 Timestamp Assignment 61
4.16 Carrier Stream Creation 62
4.17 Streaming Manager Service Invocation 67
4.18 Client Class Diagram 70

5.1 Aggregation Strategy 73
5.2 Retransmission Timeout Design 77
5.3 Delivery Strategy Comparison 78
5.4 Retransmission Strategy vs. Packet Loss 81

List of Tables

4.1 Protocol Comparison 37
4.2 Sensors in this Work 60

Chapter 1

Introduction

Nowadays a number of enabling technologies are coming together. Above all is the progressive reduction of the cost and scale of computing devices equipped with wireless communication and sensing capabilities. Sensing capabilities include cameras, microphones, GPS receivers and many different scalar sensors. Open-sourced, general-purpose and easily programmable operating systems such as Android are spreading as a result of market forces. These factors create a multitude of devices interconnected on a global scale by the Internet. Wireless access to the Internet is becoming more and more powerful. Long Term Evolution, which at time of writing is being deployed in major cities, brings significant performance advancements especially in terms of transmission delay. The Internet itself, with the deployment of IPv6, is adapting to this scenario.

This ecosystem enables so-called context-aware services. These are applications that make use of the device's physical context, possibly sending it in real-time to the Web. Examples of context-aware applications can be found in e-health, gaming and multimedia entertainment. E-health applications give doctors the ability to remotely monitor heath conditions using the patient's smartphone as a platform for collecting medical data. In the gaming and entertainment field, new creative concepts are possible if the players can insert reality in the game simply by means of their smartphone.

A common denominator of live applications is the need for timeliness. On top

of that, Quality of Experience can be declined in different ways. The focus of medical monitoring systems, for instance, might be the reliability of the transmission. Communication continuity might instead be the key quality aspect of an highly interactive gaming session. Despite the differences, it is clear that service designs should take Quality of Experience as their paramount goal and should provide a platform for its maximization.

The foundation of this class of applications borrows from heterogeneous domains: constrained environments, the Web and real-time multimedia systems. Constrained environments are made of embedded devices that have scarce resources in terms of processing power, memory and energy, interconnected by low-power and lossy networks. Wireless Sensor Networks are a typical example. Software must keep the computational and communication load to a minimum in order to enhance the device's lifetime. An handheld device, being battery-powered, has similar needs. Even though users are adapted to charging mobile phones on a daily basis, persistent sensing and wireless transmission may drain lifetime of the device below reasonable levels. Another important aspect is the integration with the existing Web. Clients access Web services using an established set of paradigms, such as Representational State Transfer. New services, also those on embedded devices, should expose the same interfaces if they want to leverage the existing infrastructure and designers know-how. Furthermore, applications that deal with evolving physical phenomena often have real-time needs. This is the case of multimedia streaming, in which the processing of information must keep up with the natural evolution of events. Such systems also need stream synchronization, that is the maintenance of the real-world temporal relation among events. Unfortunately, generally available technologies like Android and the Internet are not real-time friendly, so countermeasures must be taken.

The Constrained Application Protocol (CoAP) was created to match both the needs of constrained environments and to integrate well with the existing

Web. CoAP includes functionalities of the ubiquitous HTTP protocol, which have been re-designed accounting the low processing power and energy consumption constraints of embedded devices. At a closer inspection, CoAP is also based on the same networking layers used by current multimedia applications on the Web.

In the first phase of the work we reviewed the literature in search of suitable technologies for live, context-aware services. We investigated the different types of synchronization and challenges of real-time transport over unreliable networks, using the Real-time Transport Protocol (RTP) as reference point. We analyzed different application protocols, focusing on CoAP in particular. We motivate the choice by discussing its strengths and shortcomings by drawing a parallel with RTP. The second phase aimed at building a proof of concept on the suitability of CoAP for this type of services. We suggest a possible usage of CoAP for real-time systems that need stream synchronization. We implement a prototype to stream sensor data from an Android source to a client on the Web. It includes a CoAP streaming server developed with Android Native Development Kit (NDK) and a CoAP client written in Java. We implement an algorithm to synchronize the presentation a generic set of multimedia streams at the client, and try to characterize sender-originated asynchrony due to the non-real-time nature of data sampling. In the third phase we tried to leverage error correction mechanisms to improve Quality of Experience. We identify a set of performance metrics to compare different mechanisms, testing the prototype on a simulated network with Internet characteristics.

Chapter 2

Synchronization and Real-time Transport

2.1 Multimedia Synchronization

A multimedia system typically captures, transmits and plays a set of media streams. A *stream* is a temporal sequence of media information. Media information is for example audio or video or, as in this work, scalar sensor measurements. A stream is created by capturing a physical phenomenon over time via sensors, for instance a microphone or an accelerometer sensor, at a source node. There is often the need to transmit the stream over a network to a sink node for analysis or presentation. The stream semantic is preserved only if the presentation is correct according to the time domain [3].

A stream can be subdivided in *information units*. A continuous media object

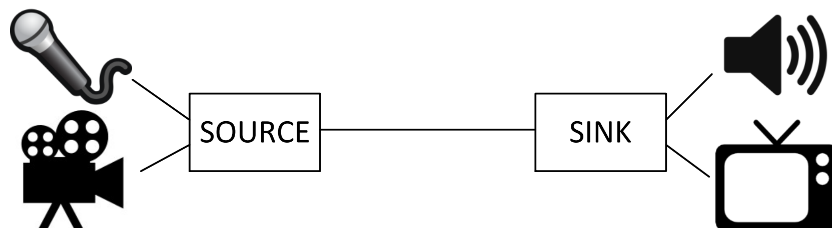

Figure 2.1: *Live multimedia system with no intermediate long-term storage.*

like audio or video is presented reproducing the sequence of information units. The temporal relations between units of a stream originate during capture and they must be preserved when the stream is presented or analyzed. Each information unit is assigned a *playout time* that should reflect the relative position of the unit within the stream. An information unit has the characteristic that each of its sub-components share a common playout time [27]. In this work an information unit corresponds to a sensor sample, so we will use the two words interchangeably.

A *synchronization unit* consists of two or more information units that, as a group, share a common interval between generation and playout of each part of the group [25]. In our context, a stream corresponds to a synchronization unit.

The synchronization problem consists in recreating on the playout side the temporal organization of information units occurring at the time of their capture or creation. In other words, the objective is to correctly specify and transfer information units capture times and to assign playout times accordingly [34]. In this work we will consider the so-called *live synchronization* problem, which implies no intermediate long-term storage between source and sink.

Asynchrony is caused by the variable processing delays that information units experience along the chain from the source to the sink nodes. The amount of variation in processing time is called *jitter*. Jitter destroys the initial temporal organization of information units. If all information units experienced the same processing delay, the stream arriving at the sink would be identical to the one at the source, only shifted forward in time.

In fact, regardless of jitter, the multimedia processing chain introduces an *end-to-end delay* (alternatively *end-to-end latency*). In this work, the system that generates most of the end-to-end delay and jitter is the Internet. The acceptable upper bound for end-to-end delay from capture to playout strictly depends on the application domain. For example the acceptable delay for interactive sessions such as video conferencing is much lower than that of one-way multimedia streaming,

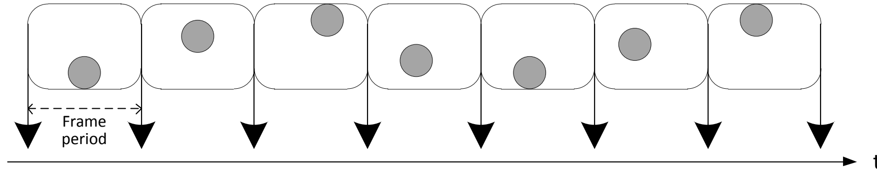

Figure 2.2: *Intra-stream synchronization between frames of a video sequence showing a jumping ball [3].*

despite being both live application. The International Telecommunication Union states that an end-to-end delay below 150 ms guarantees high quality of experience for most applications [32].

In a complex system, synchronization can be established at different levels: intra-stream synchronization refers to the time relations existing among the units of one media stream. In the simplest case, units have a fixed time validity and shall be presented one after the other in the correct order, as exemplified in Figure 2.2.

Inter-stream synchronization instead refers to synchronization of a set of continuous medias that are to be presented together. The need for inter-stream synchronization arises when the content of two media streams is related. An example is the so called lip-sync problem, where voice audio must be played synchronously to the movement of the person's lips in the video [6]. To perform inter-stream synchronization it is useful to refer to a *master stream*, which can be thought of as an orchestra director. The playout progress of the master stream is independent. The progress of all other streams in the synchronization domain, which are called *slave streams*, depends on the master. In case of lip-sync it is common to select audio as the master stream. Note that the master stream need not be an actual media stream, as an orchestra director is indeed not. It can be any independent time base. Selecting an actual stream as the master is similar to an orchestra that, instead of following the director, follows the concertmaster ("first" violin player).

Synchronization can be established at any point of the system, not necessarily

at the endpoints. This is the case, for example, of video conferencing intermediary systems that are in charge of mixing many video and audio streams into a single one. However, the best place to perform synchronization is right before the media playout at the sink node, in order to minimize the remaining jitter sources.

2.1.1 Streams and Sampling Frequency

In the simplest case, all information units in a stream have the same temporal validity. An example is a sensor that measures temperature at 1 Hz. In other cases information units have a variable temporal validity, either because the sampling frequency changes over time or because the content of a unit influences also other units. Examples are many of the common video encoders, where interframes may depend on information from other video frames [6]. An interframe is normally needed to correctly decode more than one normal frame. A stream for which the duration of each information unit is known in advance is called a *known-frequency* stream.

It may also happen that the duration of an information unit is unknown. These create an *unknown-frequency* stream, which is essentially a sequence of random events. An example is a free-spot parking counter. Consider information units to be the updates on the number of free parking spots. The latest free spot count shall be displayed until a new car enters the parking, thus for an unknown period.

The main difference between the two classes is that in known-frequency streams both the capture start time and the capture end time are known, while in unknown-frequency streams only the capture start time is known. As a consequence, a device presenting an unknown-frequency stream is able to assign a playout start time to every information unit but not a playout end time. After an unknown-duration unit has been sent to playout, it is not possible to tell whether it is still valid or not until a newer one arrives. This prevents, for instance, the insertion of silence periods when information units are lost or arrive late. It can be argued that any unknown-frequency stream can be converted into a known-frequency stream

by issuing replicas of the latest sample at a constant rate. In the following, the analysis is restricted to known-frequency streams.

2.1.2 Synchronization Metrics

Measuring intra-stream synchronization means quantifying how much jitter is reflected in the presentation of samples. Tolerance to jitter in sample presentation is determined by several factors, including the sample rate, encoding and target application. For example, TV quality video is generally unaffected by up to 10 ms jitter [30]. Other metrics proposed in the literature assume that jitter only causes presentation gaps by preventing samples to be available before their playout time. They assess intra-stream synchronization by gap probability or gap frequency measures [18].

In the context of inter-stream synchronization, there exist an objective quality metric called *skew*. The skew is the presentation time difference between two information units that occurred at the same instant in the real world. Limits of acceptable skew in the case of lip-sync have been measured interviewing humans watching video clips. Acceptable skew for this setup is ±160ms [31]. Note that presentation might not just happen to human eyes. In machine-to-machine contexts the "presentation" is consumed by a controller that has potentially much higher time sensitivity than human eyes. The stream content might even be the basis of a control loop for a safety critical system. Thus the maximum acceptable skew depends on the application domain. In our setting the stream information units are sensor samples. We seek to transmit more than one sensor stream, so both intra-stream synchronization and inter-stream synchronization are required. To our knowledge, there is no study on the skew upper bounds for presentation of context information, and this aspect is out of the scope of this work.

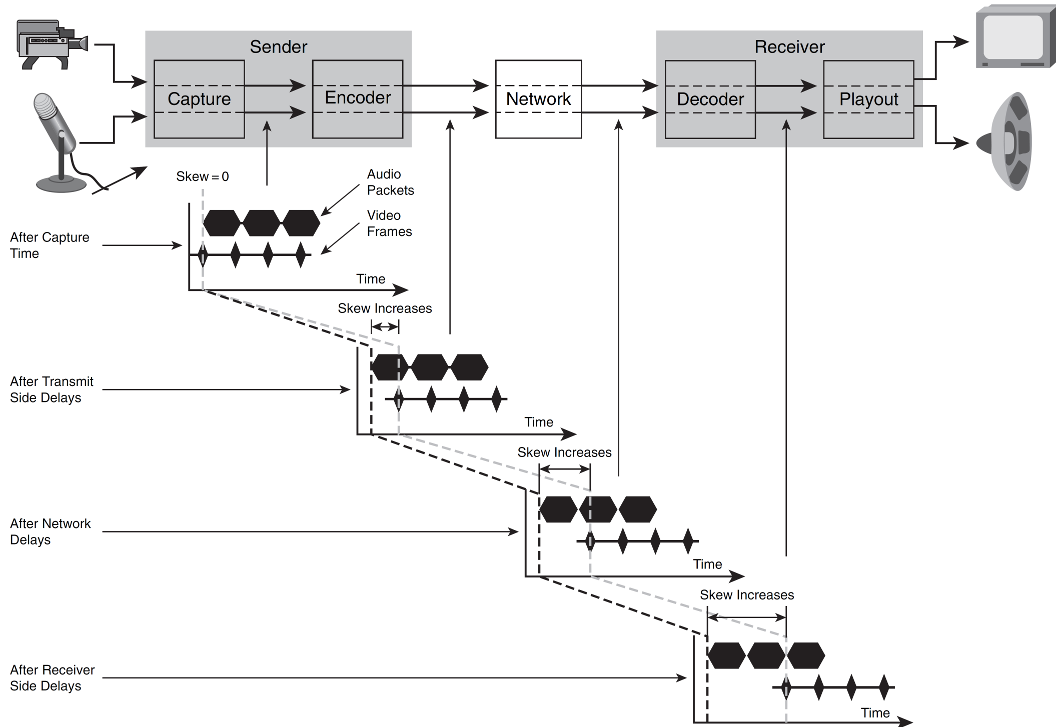

Figure 2.3: *Audio and video skew accumulation [6].*

2.1.3 Poor-man Synchronization

The simplest method that a sink node can use to synchronize a multimedia stream is to assume that information units arrive at the exact instant in which they are to be played. An information unit is played as soon as it is available at the presentation node. This technique is called *poor-man synchronization* [6]. It basically assumes "perfect" underlying systems that do not introduce processing jitter.

However naive this approach may seem, it is viable in services where the underlying technologies guarantee a certain Quality of Service (QoS) in terms of processing time. Because processing at any subsystem completes in a guaranteed maximum interval, the offset between capture and presentation of every information unit is the same. Therefore services built on hard-real time operating

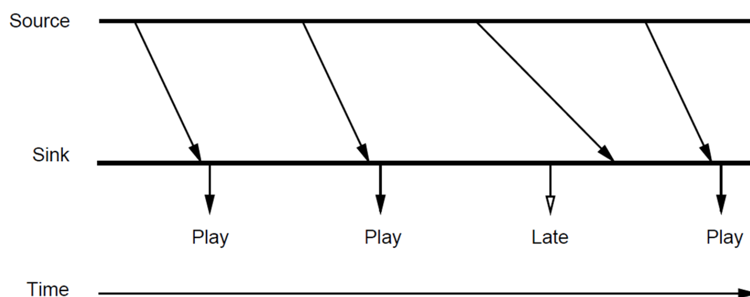

Figure 2.4: *Late information unit arrival [30].*

systems and circuit-switched networks need not take particular care to the media synchronization problem. What can still be minimized in such services to obtain even higher QoE is the end-to-end delay, which is an important requirement for interactive real-time applications (e.g. video conferencing).

However, we require our service to work on QoS-unaware systems. We assume that the software at the endpoints in charge of capturing, transmitting and playing the streams executes non-deterministically. Furthermore the Internet is by design a best-effort network. In our setting any point between the source and sink node may cause synchronization loss. This is expressed in Figure 2.4, where the third information unit experiences an unpredictable processing delay that prevents correct stream presentation. If all packets had the same delay, or at least a guaranteed upper bound, by introducing proper buffering we could ensure intra-stream synchronization simply with the poor-man mechanism, without resorting to synchronization metadata.

2.1.4 Metadata-based Synchronization

A more ingenious approach to synchronization consists in specifying the temporal relations between each information unit. The resulting metadata is called *stream specification*. To minimize the chances of asynchrony, the best place to originate the stream specification is as close as possible to the capturing hardware. Simi-

larly, the best place to interpret the stream specification is as close as possible to the presentation hardware. If the units are to be transmitted to another machine for presentation, the stream specification must be transmitted as well. For simplicity from now on we consider a stream to be self-describing i.e. embodying both the media content and the metadata needed for intra-stream synchronization.

Metadata-based synchronization must be used when dealing with temporally unreliable systems, as in this work. Therefore we need tools to formalize and transport timing metadata. The source must express the temporal relation between samples in a formal way, the channel must transport it to the sink node along with the media content and the sink node must interpret and enforce it during playout.

From the point of view of intra-stream synchronization, this is accomplished using *timestamps*. Each sample is assigned a timestamp that represents its sampling instant. Ideally, the process of timestamping is instantaneous or takes a fixed amount of time. The timestamp is sent to the sink along with the sample. The task of the sink node is to correctly interpret timestamps to reconstruct the stream timing.

For known-frequency streams, assuming that the synchronizer is aware of the sampling frequency, the timestamp can be a simple sequence number. On the other hand, samples in unknown-frequency streams can take any position in time, so the timestamp must be a clock reading. The timestamp clock rate depends on the required resolution. The lower the timestamp clock frequency, the higher the error in identifying real time instants.

Regarding inter-stream synchronization, two approaches are viable: mixing all streams into one stream through recoding at the source; ensuring that timestamps across different streams refer to a common time reference. The first approach basically simplifies the problem to an intra-stream one. It has the relevant drawback of creating stream interdependence. Data loss or errors in the elaboration of one

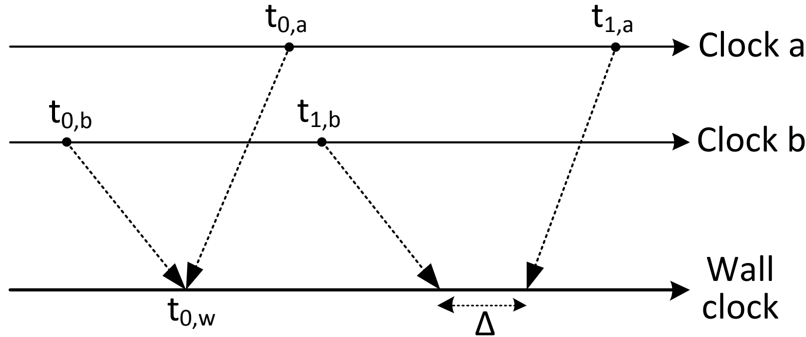

Figure 2.5: *The concept of wall clock.* $t_{0,a}$ *is the same instant as* $t_{0,b}$*, while* $t_{1,b}$ *is* Δ *seconds before* $t_{1,a}$

stream or in its transmission infect also the other mixed streams. Furthermore, intermediaries that wish to insert a media source into the synchronization domain need to perform, in most cases, complex recoding. While this may seem acceptable in case of audio-video as usually one is meaningless without the other, in the case of sensor data each stream has its own distinguished meaning (for example it is meaningful to know acceleration even if the temperature stream is faulty).

A better approach is to keep streams isolated and let them specify their own intra-stream timing. In general, each stream can have an independent timestamp reference. This means that two samples captured at the same time in two different streams might have different timestamps. Thus there is the need to relate independent timestamps to a common reference, called the *wall clock*. There is exactly one wall clock in each synchronization domain. The concept is shown in Figure 2.5.

Regardless of its offset in time or frequency, the wall clock is actually a "copy" of real time at the source. The wall clock timestamp is a valid timestamp for the sample (instantaneously assigned at time of capture), and events that happen simultaneously are mapped to the same wall clock time. Note that the intra-stream ordering of samples according to original timestamps holds also in wall

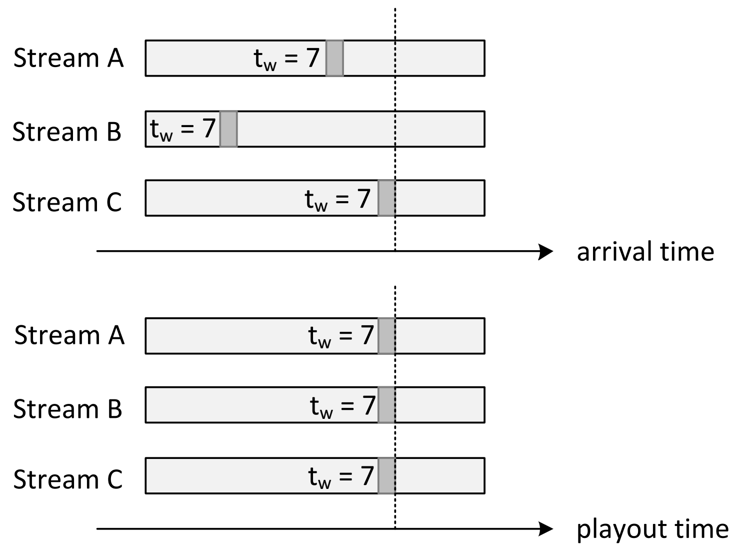

Figure 2.6: *Inter-stream synchronization. The sink aligns samples for playout based on their wall clock timestamp t_w. Shaded samples are generated at the same instant in real time, thus have the same t_w.*

clock terms. Therefore the sink can perform inter-stream synchronization by sending to playout simultaneously all the samples with equal wall clock timestamp, as shown in Figure 2.6.

A relevant drawback of the wall clock method arises in multi-source scenarios because the wall clock must be synchronized across source devices. A common solution is the use of clock synchronization protocols such as NTP or GPS for higher precision.

When using timestamp-based synchronization it is possible to divide the problem in two sub-problems. One part is the operations that occur in the end systems, from capture of the physical quantity to timestamping and from the playout decision to the actual appearance to the consumer. The other part is the operations, such as transmission over a channel, that occur while a sample is assigned a timestamp. We refer to the latter part as *timestamp zone*. Its distinguishing factor is that, within this zone, the sample position in time is uniquely identified by the timestamp, which is a simple numerical value.

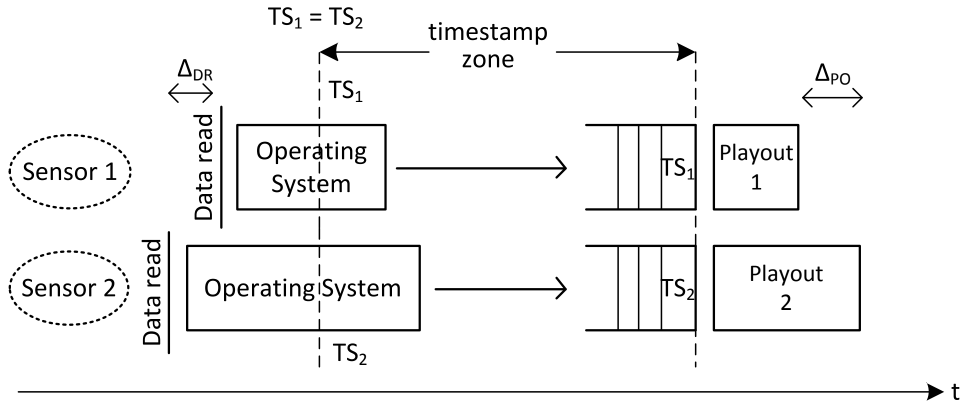

Figure 2.7: *The timestamp zone. Square boxes are operating system tasks on the same device. Total skew* $= \Delta_{DR} + \Delta_{PO}$.

Once a sample is timestamped, optimal algorithms can perform any kind of synchronization. It becomes clear that the origin of playout skew is to be found in the operations outside the timestamp zone, such as the scheduling of the tasks that handle sensor hardware or playout hardware. Moreover, this concepts suggest an architectural separation of concerns: algorithms used within the timestamp zone are platform independent, while management of capture and playout devices depends on the characteristics of the hardware and the operating systems on the endpoints.

This work focuses on synchronization on the timestamp boundary. This is driven by two facts: sample loss and jitter is much higher within the timestamp zone than outside, so this problem shall be solved first; being platform-independent, the solution can be applied to a variety of systems and platforms.

2.2 Real-time Transport over a Network

A real-time system is a system that provides guarantees or makes effort to complete certain *tasks* before a *deadline*. Multimedia systems are often real-time because of the evolving nature of physical phenomena.

This work attempts to build a real-time service. The task is the delivery of sensor samples to the sink node for presentation. The deadline is their playout time. The ideal result is that all samples be delivered to the sink node before their scheduled playout time. To measure the success rate of the system a proper metric is the Delay Constrained Reliability (DCR) [9]:

$$\text{Delay Constrained Reliability (DCR)} = \frac{\text{samples arrived before deadline}}{\text{total samples sent}}$$

This metric differs from conventional reliability metrics because it accounts for the deadline. In a real-time system, a sample that misses the deadline is as useless as a lost one. DCR as it is here defined treats all the samples as equally important. Although it is appropriate for this work, it might not always be so, for example in a prioritized real-time system where the timely arrival of some class of samples is more important than others.

An hard real-time system guarantees DCR=1, meaning that all samples sent arrive at destination before the deadline. In this work, the main impediment to the ideal DCR=1 is posed by the communication channel used, the Internet. The Internet is a best-effort network which does not guarantee delivery within an end-to-end delay bound. Travel time variations and packet loss are caused by route changes, network congestion etc. Therefore a service built on top of the Internet can only be soft real-time, that is to use mechanisms aimed at maximizing DCR [1]. In this work we use three mechanisms: UDP as transport layer protocol, elastic buffers and error correction.

2.2.1 Delivery Deadline

In the context of this work, it is useful to characterize the performance of the network in terms of three indicators:

packet loss greater than zero in best-effort networks like the Internet

average one-way transmission delay (network delay) d_N assuming a symmetric link, the round-trip time is $RTT = 2\,d_N$.

jitter (network delay variation) unpredictable and unbounded

Note that network throughput is not meaningful here because the amount of data to transmit is low. It only consists of scalar sensor measurements and their metadata, so we assume that even a low-capacity channel does not pose additional challenges. In a complete multimedia system, network throughput should instead be taken into account as video and audio streams generate significant traffic.

The delivery deadline must comply to the maximum end-to-end delay constraint. On the other hand, it is meaningless to push the deadline below the average one-way transmission delay, otherwise no samples could respect it:

$$t_c + d_N \leq \text{ delivery deadline } \leq t_c + d_{EE}$$

where d_{EE} is the maximum end-to-end delay and t_c the sample capture time. Intuitively, the later the deadline, the higher are the chances to perform the task before its expiration. The optimal choice within the constraints is therefore:

$$\text{delivery deadline } = \text{playout time} = t_c + d_{EE}$$

Note that this definition does not account for the sample duration. Alternatively, one could consider a sample anyway "useful" if it arrives before its playout end time, because it would be played for at least a fraction of its period.

2.2.2 Elastic Buffer

Thus each sample can take at most d_{EE} to arrive at destination. If $d_N < d_{EE}$, than there is some spare time that can be exploited by efforts to maximize DCR. This spare time is called *buffering delay* d_b, because in this interval samples are stored by the sink in an *elastic buffer* (or *jitter buffer*) [30].

$$d_N + d_b = d_{EE}$$

Buffering is useful for two reasons. First, it absorbs network jitter. The actual network delay can deviate up to d_b and still deliver a sample on time for playout.

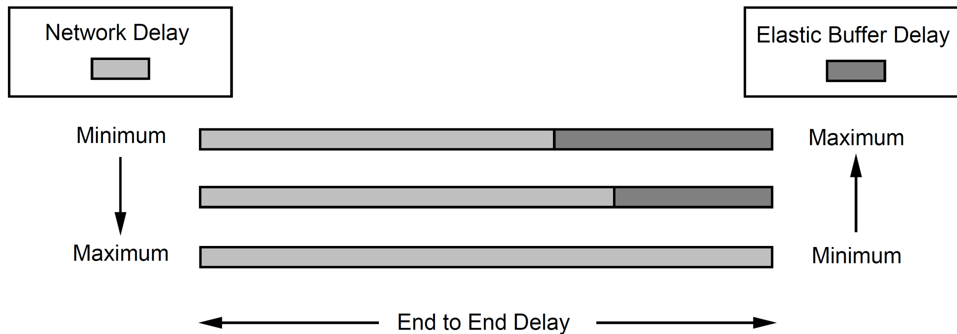

Figure 2.8: *Elastic buffer principle [30].*

Second, it opens a window of opportunity for error correction mechanisms to operate. In fact, even in an ideal case where jitter is absent, some packets could still be lost. Buffering may occur at any point along the transmission and the total buffered time is given by the sum of its parts. The optimal choice is to keep all buffering at the sink, where any subsequent jitter source prior to presentation is at a minimum [30].

2.2.3 UDP Transport

When designing a real-time service over a distributed system, one is faced with the choice of a transport protocol. The task of a real-time friendly transport protocol in the Internet is to maximize the on-time delivery effort. This implies a tradeoff between timely delivery and successful delivery. Similarly, there is also a tradeoff between overhead and successful delivery.

The two most common transport protocols in the Internet are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). Both provide multiplexing to applications on the same IP endpoint. Besides multiplexing, UDP adds little more to the raw IP service. Packets can be lost, reordered and duplicated. UDP ensures that each packet experiences the minimum one-way transmission delay offered by the network, with no attempts to recover errors.

On the other hand, TCP guarantees in-order successful delivery. If an error occurs on a data chunk, the delivery of following data chunks is delayed until the error is recovered. The result is an actual end-to-end transmission delay higher than the minimum. For real-time services this is a significant weakness because loss or jitter of one information unit reflects on the following ones, potentially causing a chain of deadline misses.

The balance between timeliness and reliability is largely dependent on the application needs. Non-real-time applications such as file transfer must achieve full end-to-end reliability but do not impose any completion deadline, so they typically use TCP. On the other hand multimedia streaming like VoIP or video conferencing accept some degree of error rate if this improves quick delivery of the media stream. UDP is the protocol of choice for these applications. UDP is also the basis of the standard Real-time Transport Protocol (RTP) [27]. Flexibility is enhanced because upper layers are free to build custom reliability services on top of UDP, for instance without imposing in-order delivery or applying error correction only to selected packets. An example is the Constrained Application Protocol (CoAP) which is described and used in the next chapters.

2.2.4 Error Correction

Sender-based error correction in unreliable networks implies the use of redundancy against errors caused by packet loss or excessive jitter. Techniques may be split in two major classes: *active retransmission* (or *backward error correction*, BEC) and *forward error correction* (FEC) [21]. Active retransmission is a reactive approach that consists in expecting an acknowledgment of reception of the packets. For any packet received, the destination must send an acknowledgment back to the sender. If the acknowledgment is not received within a *retransmission timeout* t_{RTO} then the respective packet is retransmitted.

FEC uses redundancy proactively instead of reactively. Information, however encoded, is always sent more than once to the receiver. This increases the chance

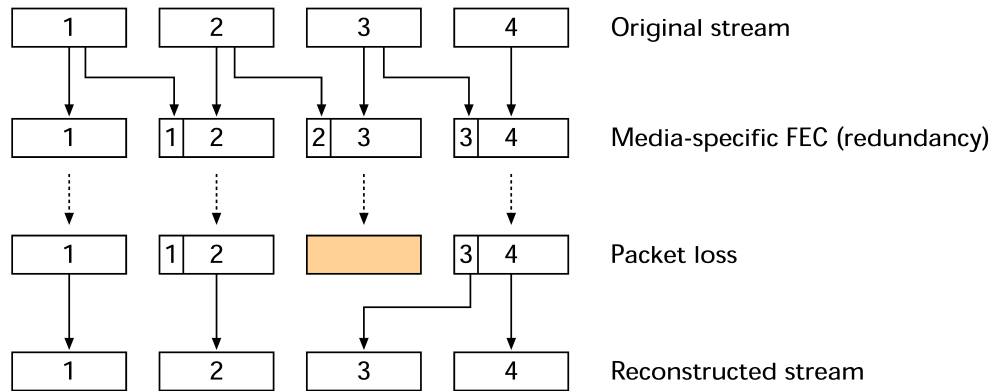

Figure 2.9: *Forward Error Correction by repetition [21].*

of receiving data. Moreover it does not require feedback from the receiver. FEC can be employed at various levels, notably at the packet level and at bit level (channel coding). We focus our attention at the packet level, as it is the layer directly under our control. A simple technique is that of *repetition*, which belongs to *media-specific FEC* [21]. It consists in transmitting each sample in multiple packets. If a packet is lost, another packet containing the same sample will be able to cover the loss, as illustrated in Figure 2.9.

The main difference between FEC and BEC lies in the selectivity. In general FEC performs better than retransmission in high loss environments because of the higher utilization of the redundant information, which is sent in any case. Also, it does not adapt to changing network conditions. On the contrary, the selective nature of BEC makes it suitable for low loss environments [21].

In a real-time service, both techniques must account for the delivery deadline. As shown in Figure 2.10, it is meaningless to send redundant information too late after the original capture because it will not arrive to destination on time.

In BEC, it is not efficient to retransmit information before $t_c + RTT$ because even success acknowledgments can not return by that time. However, pushing t_{RTO} below this threshold does increase the effectiveness of the technique because

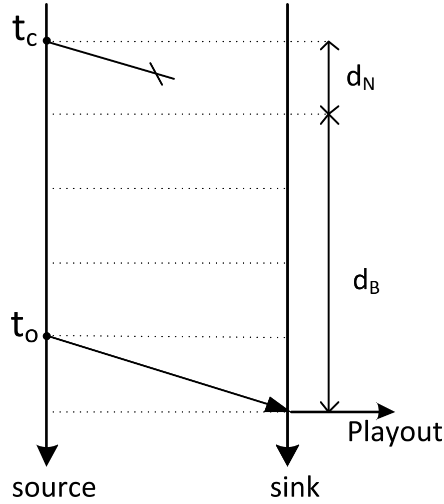

Figure 2.10: *Error correction window of opportunity.*

retransmissions tolerate more jitter. This behavior appears evident from the measurements described in later chapters. However, the retransmission timeout has an upper bound:

$$t_{RTO} \leq t_c + d_b$$

In repetition, the benefit of the scheme depends on the sampling frequency $\lambda = \frac{1}{\tau}$. A packet is issued every time a new sample appears and contains the current sample plus the last one's duplicate. The upper bound of the sampling period to operate within the window of opportunity is:

$$\tau \leq d_b$$

Similarly to retransmission, the higher the sampling frequency the higher the tolerance to jitter of the duplicate sample.

Chapter 3

Application Protocols

The goal of this work is to send in real-time a set of sensor measurement streams over the Internet and to synchronize them temporally at the receiver. In order to accomplish this task, one is faced with the choice of a suitable application protocol. Three possible alternatives are identified that may suit these needs: the Real-time Transport Protocol (RTP), the Constrained Application Protocol (CoAP) and the Message Queue Telemetry Transport protocol (MQTT). They conceptually span from the session to the application layers of the ISO/OSI model. RTP is included because it is the reference standard for real-time applications. CoAP is interesting because it has the same transport layer as RTP but it is a rich application protocol that targets constrained, machine-to-machine environments. MQTT is interesting because it proposes an alternative approach to both the transport layer and application layer of RTP and CoAP.

Following is an overview of the three protocols. Chapter 4 hosts a comparative analysis according to a series of requirements peculiar to this work.

3.1 Real-time Transport Protocol (RTP)

The Real-time Transport Protocol (RTP) [27] is currently the reference standard for real-time multimedia transport over IP networks. VoIP and video conferencing are the most natural applications of RTP. It is an IETF and ITU standard, first

published in 1996 and it is employed in most commercial systems. A thorough inspection of the relevant services offered by RTP is given in Section 4.3.1.

RTP is made of two parts: the RTP data transfer protocol and the RTP control protocol (RTCP). RTP provides delivery of data and strictly related metadata. RTCP runs alongside RTP and conveys overall session information and statistics. Figure 3.1 shows the RTP header.

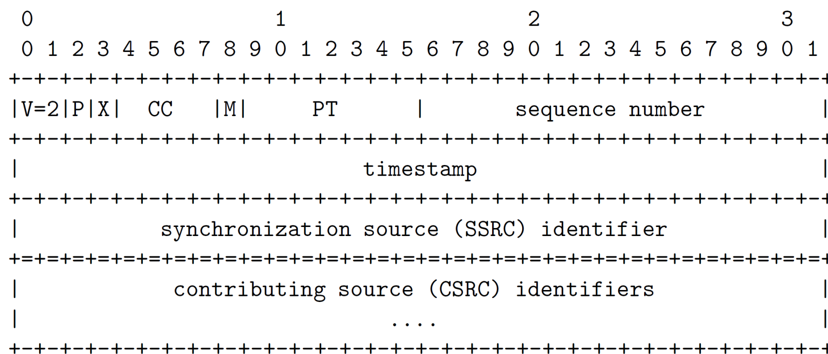

Figure 3.1: *RTP header format.*

As a quick overview of relevant fields, the *payload type* (PT) identifies the format of the payload. The *sequence number* is useful to restore the packet sequence and to detect losses. The *timestamp* reflects the sampling instant of the first media byte in the payload. *Synchronization and contributing sources* divide data chunks into groups.

RTP is deliberately not complete because it assumes that different multimedia applications have different requirements [27]. A *profile* defines an interpretation of the generic fields within the RTP specification according to the needs of an application domain. For example, a profile may define a static mapping of payload type codes to specific payload formats. The first and most common profile is the "RTP Profile for Audio and Video Conferences with Minimal Control" published in RFC3551 [26].

RTP is intended to be used in association with other signaling protocols for

Call control / Media negotiation			Lightweight sessions	Media codecs
RTSP	SIP		SAP	RTP
TCP		UDP		
IP				

Figure 3.2: *IETF multimedia protocol stack [22].*

session discovery and negotiation. Figure 3.2 outlines the IETF protocol suite for multimedia. For example the Session Initiation Protocol (SIP) is used to mark session participants, negotiate media parameters and to eventually start an RTP streaming session.

RTP is based on UDP. It does not provide full services but rather a framework where application-specific algorithms can run. This assumption, called *application-level framing*, comes from the recognition that multimedia applications are diverse in their requirements, so implementation of additional services is left as their choice [22]. For example each RTP packet contains a sequence number and each RTCP Sender Report packet contains the total number of octets sent up to that instant. Depending on their needs and complexity, receivers can use the sequence number to simply perform packet reordering or they can combine sequence numbers and the octet count to estimate packet loss and run congestion control algorithms.

RTP configures an end-to-end architecture. Intelligence resides at the end systems. Two entities are key: *senders* and *receivers*. Senders capture the media and specify its characteristics, while receivers synchronize and present the media, optionally trying to recover delivery errors. Translators and mixers are intermediaries that act as both receiver and sender. They perform tasks like stream mixing (audio and video, for example), media transcoding or protocol translation at lower layers.

3.2 Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) is a web service transfer protocol that tries to merge the interoperability of current Web service paradigms with the need for lightweight processing and communication in constrained environments. It is currently under draft at IETF [29]. At the application layer it can be thought as a compression of HTTP, as it offers the same RESTful application paradigm. However CoAP is built on assumptions typical of Internet of Things and Machine-to-machine applications: slow, unstable network and resource-constrained end systems. It aims to enable a new set of applications called Embedded Web Services [28].

CoAP is made of two sub-layers: a transaction sub-layer and an application sub-layer. CoAP is based on UDP. The transaction sub-layer adds reliability to UDP. In contrast to TCP, the reliabilty unit in CoAP is a message and not a stream, thus it simpler and more discretionary. Messages can be of four types: Confirmable, Non-confirmable, Acknowledgment and Reset. Non-confirmable messages do not imply an acknowledgment, instead Confirmable messages must be acked upon successful reception. If an ack is not received by a retransmission timeout, the message is retransmitted.

The application sub-layer implements RESTful Web services using a request/response pattern. The central idea of REST revolves around the notion of *resource* as any component of an application that needs to be used or addressed [8]. A resource is uniquely addressed by a URI. Resources can be parameterized by the URI-query portion of the URI. For example:

```
/temperature/livingroom?unit=c
```

might identify a temperature sensor in a living room that comes in Celsius degrees. In advanced interfaces even application algorithms may be modeled as parameterizable resources enabling a sort of remote programmability of the server.

Messages can be requests or responses. Requests access resources on the server in the way dictated by the request method. For example the GET method asks the server for a one-time resource representation. Responses carry the representation back to the client. An example of interaction is given in Figure 3.3 (a).

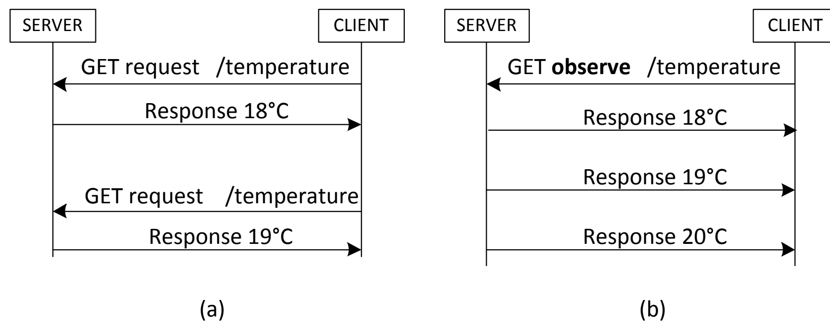

Figure 3.3: *Examples of a CoAP exchange. Classic request/response (a), observe extended (b).*

The CoAP message format is shown in Figure 3.4. The *type* (T) field indicates the message type. The *code* field indicates whether the message carries a request or a response. The *message ID* is used to match Confirmable messages with their ack and to detect deduplication. The variable-length *Options* field normally carries the resource *URI*, a *Token* used to match requests with their responses and the *Content-Format* to identify the payload type.

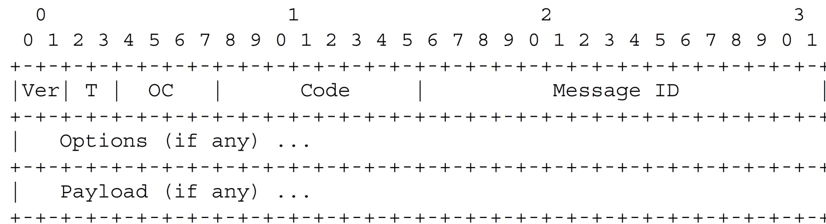

Figure 3.4: *CoAP message format.*

Standard CoAP is pull-based: to one client request corresponds only one server response. A request/response protocol is unsuitable for streaming applications.

CoAP Observe [10] extends CoAP with push notifications, so that a server can update a set of clients about a changing resource without the need of further requests, as illustrated in Figure 3.3 (b). This capability comes at a price with respect to traditional REST services: servers cannot be stateless because they must keep the observe registration state; interoperability with HTTP is no longer straightforward.

Another interesting feature of CoAP is resource discovery. Humans discover resources on the Web using hyperlinks, but this can be done by machines if standard interfaces and resource descriptions are available. CoAP servers are encouraged to provide a resource behind a well-known URI (`/.well-known/core`) that describes available resources on the server. The format is also an IETF standard, called CoRE Link Format. An example of resource discovery is given in Figure 3.5. The server informs the client in a formal way about the available video formats. The client automatically chooses the format and observes the resource with an URI-query parametrized accordingly. Thus CoAP offers an interesting set of tools to streaming applications: means for a client to request a stream from a server, capability of the server to provide the stream and the real-time friendly UDP layer underneath.

At the time of writing CoAP is being very actively researched. Most of the protocol appeal lies in the interoperability with the existing Web, since CoAP-HTTP proxing is natural (except for Observe sessions), and in the flexibility of the REST paradigm. An instructive example is given in [15]. The authors propose an Internet of Things architecture called "Thin-server Architecture". Embedded devices run Web servers based on CoAP. They only expose their sensors and actuators as a Web API and move the control firmware out from internal microcontrollers to the Web, as in Figure 3.6. Commercial products such as Cisco smart grid solutions already use CoAP [1].

[1] `http://blogs.cisco.com/ioe/beyond-mqtt-a-cisco-view-on-iot-protocols/`

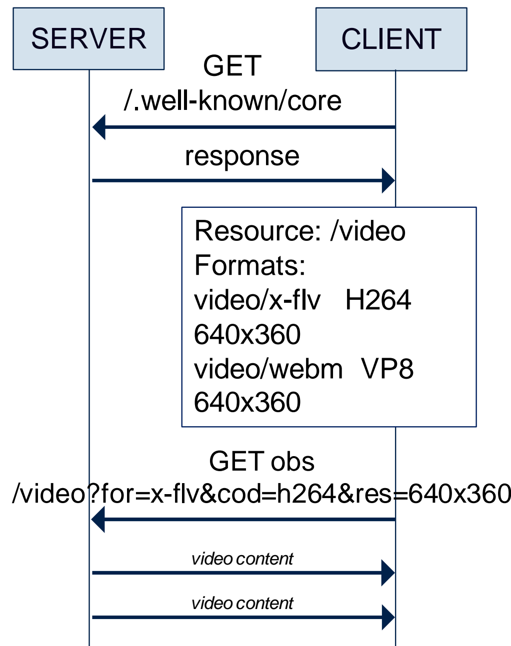

Figure 3.5: *Example of CoAP discovery useful to negotiate streaming parameters.*

3.3 Message Queue Telemetry Transport Protocol (MQTT)

The Message Queue Telemetry Transport [12] is a lightweight messaging protocol developed for telemetry data reporting. It was designed to be supported by constrained environments where resources are scarce in terms of network bandwidth, processing and power. MQTT clients footprint is roughly 30KB for a C implementation or 100KB for a Java implementation. The smallest packet size for a message is 2 bytes [17]. MQTT is open and royalty-free, born in 1999 at IBM and currently under standardization at OASIS. It is used in diverse commercial systems, notably the Facebook Messenger mobile app [2].

MQTT relies on TCP/IP for point-to-point, session oriented, auto-segmenting transport with in-order delivery [11], thus the simplicity of the protocol is compensated by a relatively complex underlying stack. It defines Quality of Service

[2] `http://mqtt.org/2011/08/mqtt-used-by-facebook-messenger`

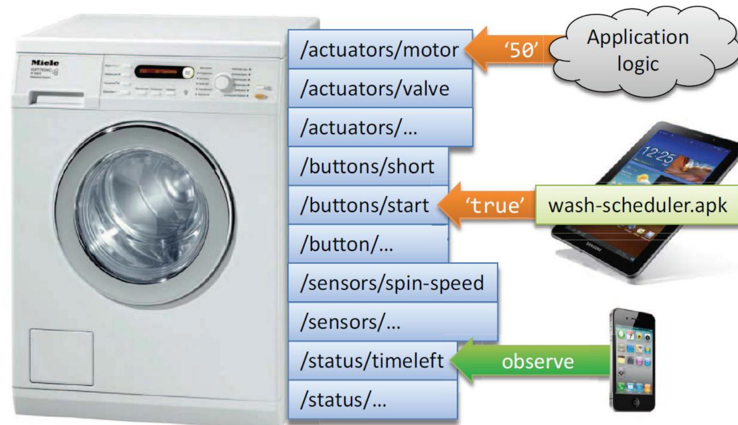

Figure 3.6: *Thin-server Architecture. A washing machine controlled from the Cloud [15].*

classes to ensure *at-most-once*, *at-least-once* or *exactly-once* message delivery to the destination making use of acks and controlled retransmissions.

MQTT features a publish/subscribe model. Three architectural elements are defined: *publishers*, *subscribers* and *message brokers*. Publications are organized in *topics*. Each publication made by a publisher refers to a particular topic. Subscribers simply request a subscription to a topic and then start receiving messages that belong to it. Message brokers serve as intermediaries, effectively making the participating entities loosely coupled. In a similar fashion as REST architectures, publishers do not need to know who are its subscribers and vice versa. The abstraction of REST resources is analogous to publish/subscribe topics.

The publish/subscribe paradigm is essentially push-based. Clients have no means to control publishers since they cannot make explicit requests. They can only decide what to receive among the available topics. In our case this is a downside in that clients who want to receive a sensor stream either accept the stream parameters or negotiate in advance via signaling mechanisms beyond MQTT scope.

MQTT defines several message types. The key ones are the PUBLISH and SUBSCRIBE messages. Other message types are used to establish the client-

broker connection, to acknowledge message reception or unsubscribe. Publishers use PUBLISH messages to push telemetry data to the network. PUBLISH messages are relayed to all subscribers who have previously shown interest to that topic via a SUBSCRIBE message. Figure 3.7 shows an example of MQTT PUBLISH message. The first two bytes are the fixed header. The message type indicates a PUBLISH message. To be properly handled a PUBLISH message must carry the topic name encoded in UTF-8. Application-specific content goes in the payload.

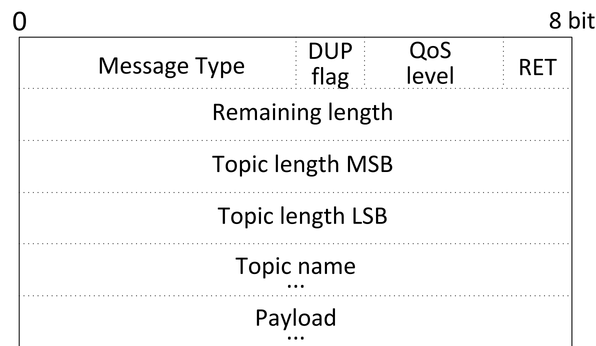

Figure 3.7: *MQTT PUBLISH message format.*

MQTT offers no support for synchronization metadata in the packet header nor it identifies the payload content type. Synchronization metadata must be included in the payload and its format must be agreed with out-of-band means.

Despite not being standardized, MQTT is quite mature. There exist free implementations for a variety of platforms and MQTT systems are sold by major vendors. However most existing services in the Web do not use MQTT. Translation is needed if one wants to connect an MQTT system to the existing Web.

Chapter 4

Design Choices and Implementation

4.1 Measuring Quality of Experience

Drawing from the discussion in Section 2.1.2, we propose a combined metric called Playout Reliability:

$$\text{Playout Reliability (PR)} = \frac{\text{samples played at their playout time}}{\text{total samples sent}}$$

An ideal service keeps $PR = 1$. By splitting PR in two different components it is possible to highlight synchronization performance and real-time transport performance:

$$\text{PR} = \frac{\text{samples played at their playout time}}{\text{samples arrived before deadline}} \times \frac{\text{samples arrived before deadline}}{\text{total samples sent}}$$

The first component is named SYNC:

$$\text{SYNC} = \frac{\text{samples played at their playout time}}{\text{samples arrived before deadline}}$$

SYNC captures synchronization performance. It measures the ability of the synchronizer to play at the proper time the samples that have arrived on time. If taken across a whole synchronization domain, it characterizes both intra and inter-stream synchronization. Note that if poor-man synchronization is used, network jitter drives SYNC<1. Using metadata-based synchronization and restricting the

analysis within the timetamp zone it is possible to achieve “perfect” SYNC=1, meaning that both presentation jitter and skew are absent.

The second component has been already defined in Section 2.2 under the name Delay Constrained Reliability (DCR). It captures real-time delivery performance in the form of gap probability. As mentioned, due to the unreliable nature of the communication channel, it is impossible to obtain DCR=1. It can only be maximized. Note that SYNC and DCR are totally independent. Efforts to improve one do not impair the other.

Two other meaningful QoE metrics are:

$$\text{Network Load} = \frac{\text{bytes exchanged}}{\text{total samples sent}}$$

$$\text{Packets per Sample (PPS)} = \frac{\text{packets exchanged}}{\text{total samples sent}}$$

Bytes and packets exchanged are measured on the source network interface. They account for UDP payload bytes and UDP packets respectively.

This set of metrics is significant and comprehensive because it captures the two main aspects of QoE. While PR measures the effectiveness of synchronization and transmission strategies, network load and PPS measure their efficiency. Also, many other QoE features that are beyond the scope of this analysis are in tight relation with network load and PPS. For example, it is well-know that battery power consumption of a mobile device mainly depends on the load of the radio circuits, both in terms of total active time (network load) and in terms of activity cycles (PPS) [5]. Note that these metrics are application-independent. They give a QoE overview for many application classes at the expense of accuracy for a specific class.

4.2 Synchronization Algorithm

This section gives a formal description of the synchronization algorithm that has been implemented at the sink device. It tackles both intra and inter-stream syn-

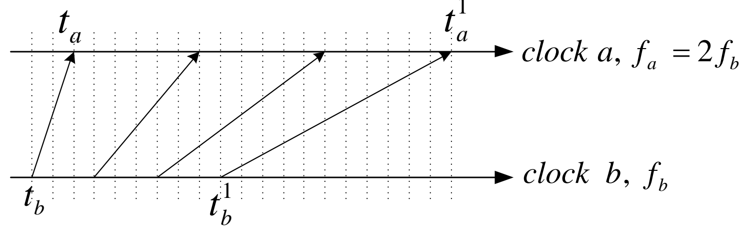

Figure 4.1: *Mapping clock instant t_b^1 on clock a. Knowledge of the clock mapping tuple is required.*

chronization. It achieves SYNC=1 on the timestamp boundary.

We define *clock mapping* as the tuple (t_a, f_a, t_b, f_b), where t_a, t_b are values of clocks a and b and represent the same real time instant. f_a and f_b are the frequencies of clocks a and b. The function $cmap(t_y, x)$ specifies how to obtain the corresponding time instant on clock x knowing an instant on clock y, as illustrated in Figure 4.1:

$$t_a^1 = cmap(t_b^1, a) = t_a + \left(\frac{f_a}{f_b}\right)\left(t_b^1 - t_b\right)$$

Note that if $f_a = f_b$ the mapping represents two clocks that are in offset $t_a - t_b$, as for example the clock of two machines that express time in nanosecond precision. If both $f_a = f_b$ and $t_a = t_b$ then a and b are in fact the same clock.

Let us consider a sequence of samples and their associated timestamps:

$$\{(S_x, TS_x)\} = \{(S_0, TS_0), (S_1, TS_1), (S_2, TS_2)\ldots\}$$

Timestamps are derived by the timestamp clock according to this policy:

- TS_0 is chosen randomly
- $f_{TS} = 1$ kHz
- $TS_{x+1} = TS_x + \frac{f_{TS}}{\lambda}$ (where λ is the sampling frequency)

thus it holds $TS_0 < TS_1 \ldots < TS_x$. At the sink, TS_x is converted to a wall clock time WTS_x and then to a playout time p_x. Received samples are inserted into an

ordered queue based on TS since it indicates the playout order, independent on the playout time that will result. The queue head holds the sample with lowest TS. The playout process is modeled as an output device that generates playout requests for a sample at a constant rate $f_{PRQ} = \frac{1}{\tau_{PRQ}}$.

Introducing inter-stream synchronization, the objective is to assign the same playout time to samples that have the same wall clock timestamp. We name WTS_x the wall clock timestamp of sample S_x. Assuming that the sink knows the mapping between the timestamp clock and the wall clock in the form (WTS_0, f_{WC}, TS_0, f_{TS}), then it can use *cmap*() for the conversion:

$$WTS_x = cmap(TS_x, wall\,clock) = WTS_0 + \left(\frac{f_{WC}}{f_{TS}}\right)(TS_x - TS_0)$$

The next step is to assign to each sample a presentation time in the receiver system clock terms. In other words:

$$p_x = p\left(WTS_x\right)$$

To compute the playout time in presence of inter-stream synchronization we use the notion of master stream and blind delay. The master is the stream that sets deadlines. This algorithm does not include a policy to alter the master stream time i.e. when it is appropriate to postpone or anticipate playout, but slave streams do follow any master time alteration.

The *blind delay* method [25] assumes that the first packet in a stream experiences the average one-way transmission delay. The playout time of the first sample in a stream is determined by adding a fixed offset to the sample arrival time a. The fixed offset is called blind delay d_b. We accept the average delay assumption. A better algorithm could estimate d_N using the first k packets and update d_b on-the-fly. This algorithm is already robust to changes in d_b. Thus the blind delay corresponds to the elastic buffer length. The playout time of the first sample of the master stream becomes:

$$p_0^m = a_0^m + d_b$$

where m indicates samples of the master stream. Let us assume for the moment that there is not going to be any pause or skip in the presentation of the master stream. Let us also assume that the rate of the receiver system clock is the same as the wall clock, $f_w = f_{rx}$. Then the playout time of any other sample is decided:

$$p_x = p_0^m + (WTS_x - WTS_0^m) = WTS_x + (p_0^m - WTS_0^m)$$

We define the quantity $(p_0^m - WTS_0^m)$ as the Master Playout Offset (MPO). In other words:

$$MPO = p_0^m - WTS_0^m = (a_0^m + d_b) - WTS_0^m$$

defines a mapping between the wall clock and the receiver clock. MPO can simply be added to WTS of any sample to directly obtain its playout time. If $f_w \neq f_{rx}$ this simplification is no longer valid, but the mapping still holds using the full *cmap*() function. Note that MPO embodies the relation between two unsynchronized clocks: the streams' wall clock and the receiver system clock. Under these assumptions, synchronization is performed by Algorithm 1, which is executed at τ_{PRQ} intervals.

Algorithm 1 Playout algorithm, static version

t = now()
if $t - \frac{\tau_{PRQ}}{2} < p_x < t + \frac{\tau_{PRQ}}{2}$ **then**
  play S_x
  x=x+1
else if $p_x < t - \frac{\tau_{PRQ}}{2}$ **then**
  x=x+1
end if

If we allow the playout of the master stream to pause or skip as illustrated in Figure 4.3, then MPO changes over time. In case a sample of the master stream is late, for example, the policy could be to pause the master stream (and consequently all the slave streams) until the sample has arrived. The playout time

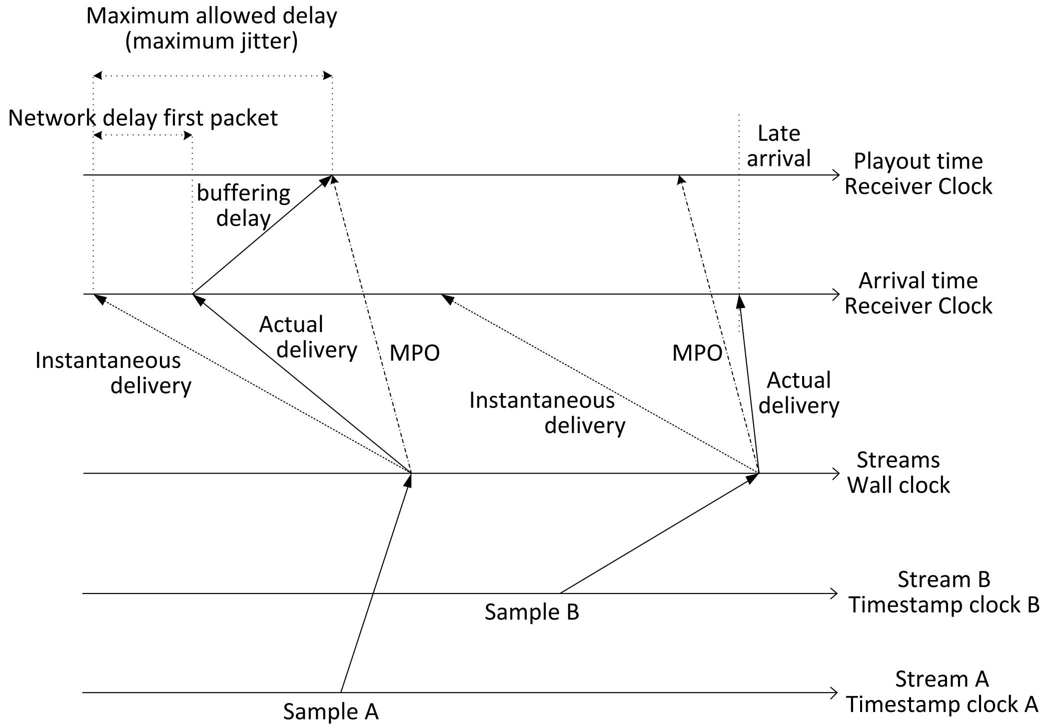

Figure 4.2: *Synchronization example. All clocks are unsynchronized. Sample A is the first received in the session.*

is shifted forward in time, and MPO must follow this behavior. Let us define the functions:

$pause\,(t)$ counts total paused time from playout start

$skip\,(t)$ counts total skipped time from playout start

The playout time of sample S_x, which now varies over time, is given by:

$$p_x\,(t, WTS_x) = p_0^m + (WTS_x - WTS_0^m) + pause\,(t) - skip\,(t)$$

and the Master Playout Offset is redefined as:

$$MPO\ = p_0^m - WTS_0^m + pause\,(t) - skip\,(t)$$

Because in case of a pause MPO increases (decreases in case of a skip), the buffer average occupancy will also increase (decrease). A possible online algorithm that

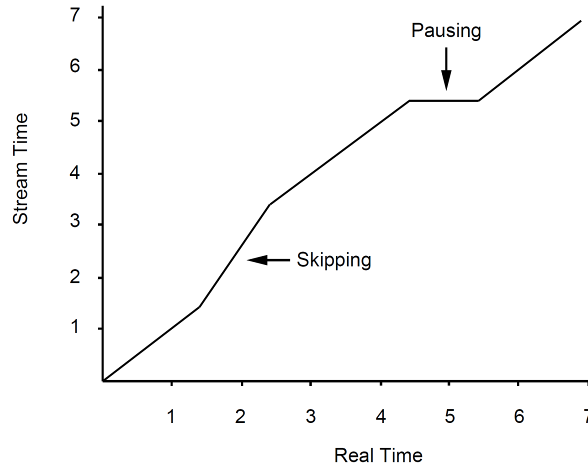

Figure 4.3: *Alterations to the presentation of the master stream, which is allowed to pause or skip samples.*

enforces inter-stream synchronization in the complete scenario is Algorithm 2. It is heavier than the previous one in that p_x must be recomputed at every iteration.

Algorithm 2 Playout algorithm, online version

t = now()
if $t - \frac{\tau_{PRQ}}{2} < p_x(t) < t + \frac{\tau_{PRQ}}{2}$ **then**
  play S_x
  x=x+1
else if $p_x(t) < t - \frac{\tau_{PRQ}}{2}$ **then**
  x=x+1
end if

4.3 CoAP as Application Protocol

Four major requirements are identified for the choice of an application protocol, all aimed at maximizing the QoE metrics given in Section 4.1.

UDP-based The protocol should be based on UDP, following the discussion in Section 2.2, in order to keep network latency to the minimum.

	RTP	CoAP	MQTT
UDP-based	yes	yes	no
Stream-friendly	yes	yes (Observe extension)	yes
Synchronization support	yes	no	no
Reliability support	no	yes	yes

Table 4.1: *Protocol comparison.*

Stream-friendly The protocol messaging pattern should naturally accommodate a directed flow of messages from one endpoint to another without excessive overhead.

Synchronization support The protocol should transport synchronization metadata in order perform synchronization at the sink.

Reliability support The protocol should support selective backward error correction as well as forward error correction in order to maximize successful delivery.

Table 4.1 shows a comparison between the protocols according to our requirements. The most influencing factor is perhaps the underlying transport protocol. It is theoretically possible to run MQTT on UDP provided that messages do not exceed one UDP packet in length and implementations do not rely on underlying layers for QoS classes. RTP and CoAP are native on UDP. CoAP has even an extension to support messages longer than one UDP payload [4].

All three are stream-friendly protocols. They include information in the headers to identify to which context the payload belongs. Even the impractical pull operation typical of request/response paradigms is spared in CoAP with the Observe extension.

Another important factor is the support for reliability. Forward error correction is always possible since it travels as data in the payload. Both MQTT and CoAP provide a native retransmission service. It is possible to build such service

on RTP but it is error prone and uncommon in current practice because video and audio encodings use forward error correction and are somewhat loss-tolerant.

Synchronization semantics is native in RTP but can be added in CoAP and MQTT with a proper payload format (or in other ways detailed below). At the extreme case, RTP packets may be encapsulated in CoAP or MQTT payloads.

We chose CoAP for our service. The only significant drawback is that it lacks synchronization fields in the header. A softer drawback is the higher complexity with respect to RTP, but it is traded for a comprehensive architectural setup plus a set of features that can largely substitute session negotiation protocols and integrate well with the existing Web. Furthermore nowadays reasearch on CoAP is very active and, coupled with the standardization efforts at IETF, it suggests a widespread penetration in real systems. Last but not least, the transport of real-time media over CoAP is a novel application with no examples found in the literature at the time of writing.

4.3.1 Introducing Synchronization Semantics in CoAP

In this chapter it is identified the essential synchronization information that a real-time protocol should carry. The analysis is bound to one use case, that is the transport of one or more streams of context information, so that non-fundamental aspects (such as fragmentation of large media chunks) are not considered. RTP is taken as the reference due to its on-purpose design for this matter and its widespread acceptance. The following analysis reflects the minimal subset of RTP semantics.

Stream identification

It is conveyed by the SSRC field in RTP packets. All packets with the same SSRC form part of a timing and sequence number space. A receiver can group these packets together for intra-stream synchronization.

Participant identification

This information is conveyed by the CNAME field in RTCP SDES packets. It provides a unique name for each endpoint. A participant identifier can be used to associate multiple media streams (SSRCs) from the same source endpoint, thus making possible inter-stream synchronization [22].

Sequence conservation

Represented by means of a monotonically increasing sequence number in RTP packets. It is used to identify packets and to recognize if packets are received out-of-order or get lost. It does not express timing information.

Timing conservation

RTP uses for this purpose two timestamps: a timestamp associated to each RTP packet which denotes the sampling instant of packet's media content. It is used to establish intra-stream synchronization. Timestamps of different streams are not synchronized. Another timestamp is included in RTCP Sender Report packets to convey the clock mapping between RTP timestamps of different SSRCs and the wall clock. The latter is useful to establish inter-stream synchronization.

Receiver feedbacks

To aid in fault isolation and performance monitoring, quality-of-service measurement support is useful. Measures of interest are packet loss, packet delay variation, clock drift et. Some of these measurements can be conducted only by the receiver endpoint and can be acted upon only by the sender. Thus a real-time protocol should support a flow of messages from receiver to sender. In RTP/RTCP, Receiver Report messages serve this purpose.

We now try to draw a parallel between standard CoAP features and the necessary synchronization semantics identified previously.

Stream identification

In a request/response protocol, a client needs to match responses it receives to the requests it had sent. To achieve this in CoAP, the server is required to replay in the response same Token value of the request. A client should choose the token randomly at each request. The Observe extension of CoAP states that the same Token must be replayed in every notification. Notifications that have the same Token belong to the same original request. From the CoAP server's viewpoint, i.e. the stream source, also the pair [client's IP/port, resource] identifies a stream. The cause is that there can be only one active Observe relation per resource per client. However, these two pieces of information do not identify a stream for the client, because notifications do not explicitly include the URI of the resource they represent. The notification-resource matching task is left to the client, indeed using the Token. We argue that CoAP's Token possesses equivalent semantics as the SSRC field in RTP and can thus be used for stream identification.

Participant identification

In principle, participant identification could simply be performed by looking at the message's source IP address. However [22] details a series of reasons why such approach should be avoided in the case of media streaming. In fact media streams could pass through intermediaries, thus hiding the original IP address. RTP uses SSRCs and CSRCs for this reason. We note that standard CoAP provides no participant identification semantics beyond what emerges from the UDP/IP layers. Let us assume that a client is unaware of the source network address of incoming messages. Then it cannot tell whether two messages that bear the same Token have originated in two different endpoints (message IDs are not helpful in this case). Due to this lack, we have to add participant identification semantics to CoAP by other means.

Sequence conservation

The Observe extension of CoAP states that the value of the Observe option, to be carried in every notification, must increment as a notification sequence number. Thus the Observe option provides sequence conservation.

Timing conservation

There is no concept of time in CoAP or its standard extensions. This information must be properly added.

Receiver feedbacks

Observe interactions in CoAP do not envisage a message flow from client to server. To overcome this problem in CoAP, two approaches may be considered. Both involve making a request to the server and using its payload (or header options) to host the Receiver Report semantics. The two approaches are:

Observe GET this approach consist in making a new Observe GET request to the resource that is already being observed. It is viable because a GET Observe request received when an equivalent observation is active causes the server to seamlessly replace the old one with the new one. Thus all equivalent Observe requests after the first one are, in practice, interpreted as one-time client-to-server messages. However this approach has a major drawback: it triggers the change of the Token (the true stream identifier), which should not change until the stream entity as a whole is destroyed.

POST this method consists in sending a POST request on the resource that is being observed. Since POST is a totally unrelated request with respect to the stream (i.e. it has a different Token), we must add the Token of the Observe relation it refers to. Since the stream's Token is untouched, the stream entity continues to exist. Of course a POST

request implies a response, which is not really needed (or at least is not envisaged by RTP), but can be useful for future extensions.

We find the POST method more suitable, both because it does not change a fundamental piece of information such as the stream identifier and because it offers a platform to carry information not yet thought of (for example, the client asking to change the sampling frequency of a sensor on-the-fly).

We outline three strategies to accommodate synchronization metadata in CoAP. Of course the use of CoAP incurs some overhead with respect to simple RTP. However this overhead might be useful to accomplish other (related) tasks.

Full RTP encapsulation

Full encapsulation is a valid strategy because one CoAP message can map to one UDP packet. The goal is to turn the CoAP layer into a simple messaging provider (as UDP would be). A possible encapsulation pattern is the following: consider two endpoints, S and C, with S being the media source. S might expose a dummy CoAP resource representing the RTP protocol entry point. They might open an Observe “channel” to associate C with the RTP resource, and use notifications to convey RTP packets from S to C. Notifications might be Non-confirmable messages, so that the only overhead incurred with respect to bare RTP is the CoAP header itself. Backwards messages from C to S might be implemented in the ways described above. The Observe GET approach is probably more suitable in this case because the Observe Token needs not distinguish streams. The encapsulation strategy has the drawback that header fields with the same meaning appear twice, so useless overhead increases. Other caveats might arise if a closer inspection of the encapsulation strategy is conducted, which is beyond the scope of this work.

CoAP header options

The CoAP standard provides a mechanism to extend the protocol with further header fields. A CoAP message header may include a list of options. An option is defined by an Option Number and an Option Value. The semantics of some option numbers is defined in the standard. A user can customarily use the non-assigned option numbers to augment the header with more information. As an example, the Observe extension to CoAP defines a new option to distinguish normal requests from Observe requests. The value of the Observe option is a sequence number that increases at each response to the client. Within this framework the desired synchronization metadata can be added to CoAP by defining new options. This approach basically makes the minimal RTP header a subset of the CoAP header.

Payload format

This approach similar to what RTP foresees. From the RTP standard: "*unlike conventional protocols in which additional functions might be accommodated by making the protocol more general or by adding an option mechanism that would require parsing, RTP is intended to be tailored through modification and or additions to the headers as needed*". Following this rationale, all the information missing in the CoAP header shall be carried in the payload.

The latter is the solution selected for the prototype, as illustrated in Figure 4.4. This choice privileges flexibility and separation of concerns. Media specification needs are very diverse and it is difficult to identify information common to all specifications. For example, consider a likely scenario where we want to bundle more than one sample per packet, then packet must carry as many timestamps as the samples. To achieve this, even if we had a header field for one timestamp, we would anyway need a payload format to layout the remaining timestamps. An instructive example of payload format defined for RTP to transport MPEG4 elementary streams can be found in [33]. Whether our choice is also the optimal

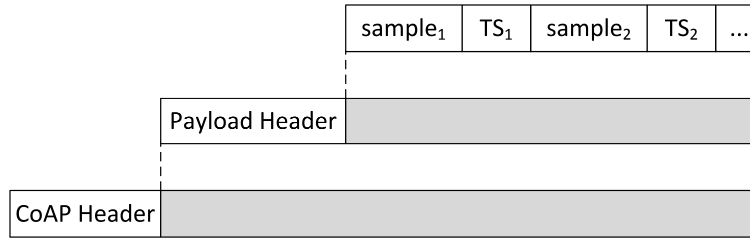

Figure 4.4: *Encapsulation of information in our CoAP extension.*

in terms of overhead should be the object of further study.

Let us follow a CoAP-based streaming session to illustrate exactly our extension. Our extension consists of: the definition of a payload format; the definition of three message types, orthogonal to CoAP's requests and responses. Our extension can be seen as a new layer above CoAP. We remark that we did not seek to optimize overhead and we targeted the extension specifically to the objective of sending sensor data. The layout of the fields and the expressiveness of our extension may not be optimal.

Observe request

Untouched

Observe notification

They form the essential part of the media stream. They include the Observe option and the Token in the CoAP header. The payload of every notification follows a well-defined format. It starts with a 4-byte header. The first byte indicates the Packet Type (PT). The semantics of all other bytes in the payload depends on PT. If the PT=1, the payload of this notification carries pure media data, equivalent to a RTP packet. If PT=2, the notification is a Sender Report (SR), mirroring the concepts of a RTCP Sender Report packet.

Data packets carry the media data (in our case the sensor measurements) along with the corresponding timestamps, which are the only piece of in-

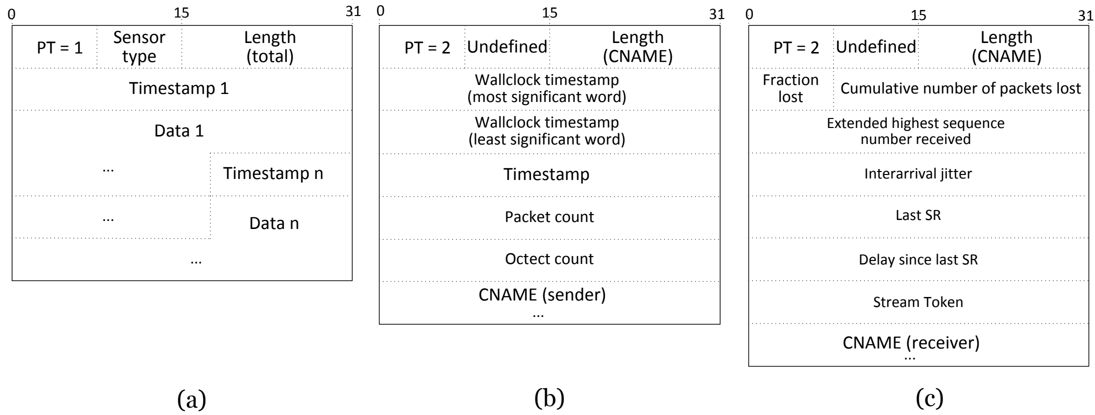

Figure 4.5: *CoAP extension payload format. Figure (a) represents Data payload, Figure (b) represents Sender Reports payload, Figure (c) represents Receiver Reports payload.*

formation missing in the CoAP header in this case. The format of sensor measurements is defined by the Sensor Type field. More than one sample of the same Sensor Type can be included in each Data packet by concatenating them and updating the length field accordingly. The payload structure for PT=1 is shown in Figure 4.5 (a).

Sender Reports are useful for inter-stream synchronization. The payload structure of a Sender Report is shown in Figure 4.5 (b). The clock mapping is given by the Wall clock timestamp and Timestamp fields. Bits that correspond to the Sensor Type field in Data packets are undefined because the format of Sender Reports is independent of the sensor. The sequence number should be increased also when sending a SR, because it is part of the Observe session. This fact however prevents the client from detecting loss in the actual media stream, as Sender Reports are not sent at regular intervals. In our implementation we chose to keep the sequence number steady in Sender Report notifications, so that the sequence number space of the actual data packets is continuous.

Receiver Feedbacks

As stated above, we have chosen to implement receiver feedbacks by means of POST requests. They are addressed to the resource generating the stream. The payload of a Receiver Report is outlined in Figure 4.5 (c) and draws from RTCP Receiver Report messages. To perform stream identification, it includes the Token of the Observe session it refers to.

4.4 System Architecture

The choice of CoAP for this service implies a specific application architecture. CoAP identifies a client and a server. In this work the server is the mobile device because it is the provider of sensor readings. The client is any machine that connects to the Internet. The architecture is depicted in Figure 4.6.

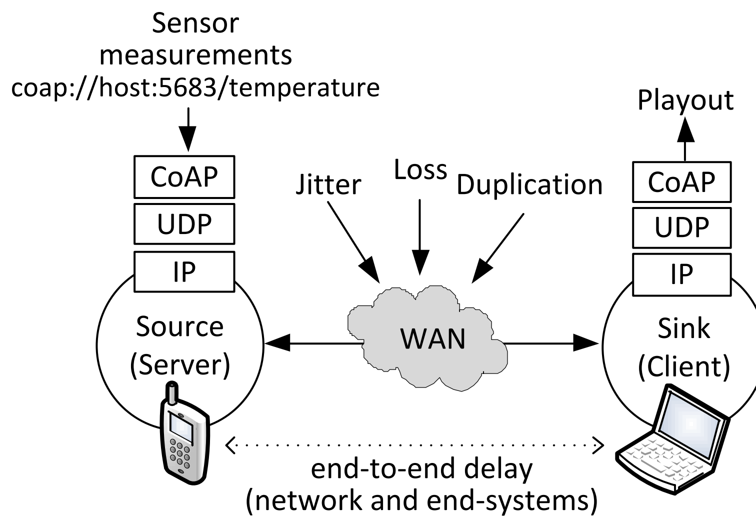

Figure 4.6: *System architecture.*

The fact that an embedded server has constrained resources shall drive the separation of functionalities between client and server. On one extreme there is a so-called fat client, in which much of the Web application logic resides. On the

other, thin clients are very simple and need a fat server in order to accomplish their tasks. In our case the basic idea is to have a thin-server architecture [15]. The server only acts as a sensor data source and provides lower-layer functionalities such as packet retransmission. Analysis and decision-making is left to the client, which may in turn send commands to the server. Receiver feedbacks identified in Section 4.3.1 serve exactly this purpose.

The server interface allows both read and write operations on resources. We use two of the four CoAP request methods: GET and POST. The GET method is used to ask a resource representation to the server, either one-time or by establishing an Observe relationship. Resource representations are carried in the body of GET responses. The client sends a POST request whenever it wants to update the status of a resource. As explained in Section 4.3.1, we use POST to report statistics in the context of an active Observe session. More advanced services might use it to modify on-the-fly parameters of the session (for instance the stream sampling frequency) or to issue generic commands to the server.

A typical exchange runs as illustrated in Figure 4.7. The clients subscribes with a GET Observe request. If the server accepts, it starts sending the resource representation (temperature readings in Figure 4.7) whenever it changes. Resources may change at regular intervals if they represent a known-frequency stream or at random intervals if they represent an unknown-frequency stream. At regular intervals the server sends a Sender Report. Also at regular intervals the client sends a Receiver Report. Each POST triggers a response, whose semantics depends on the application. Since we only send statistics via Receiver Reports, our server always replies with a positive 2.04 “Changed” code. Tokens are used by CoAP to match request and responses.

We model each sensor as a CoAP resource. Available resources are built-in in our server and reflect which sensors are installed on the device. The concept is very flexible so many other entities can be modeled as resources, for example

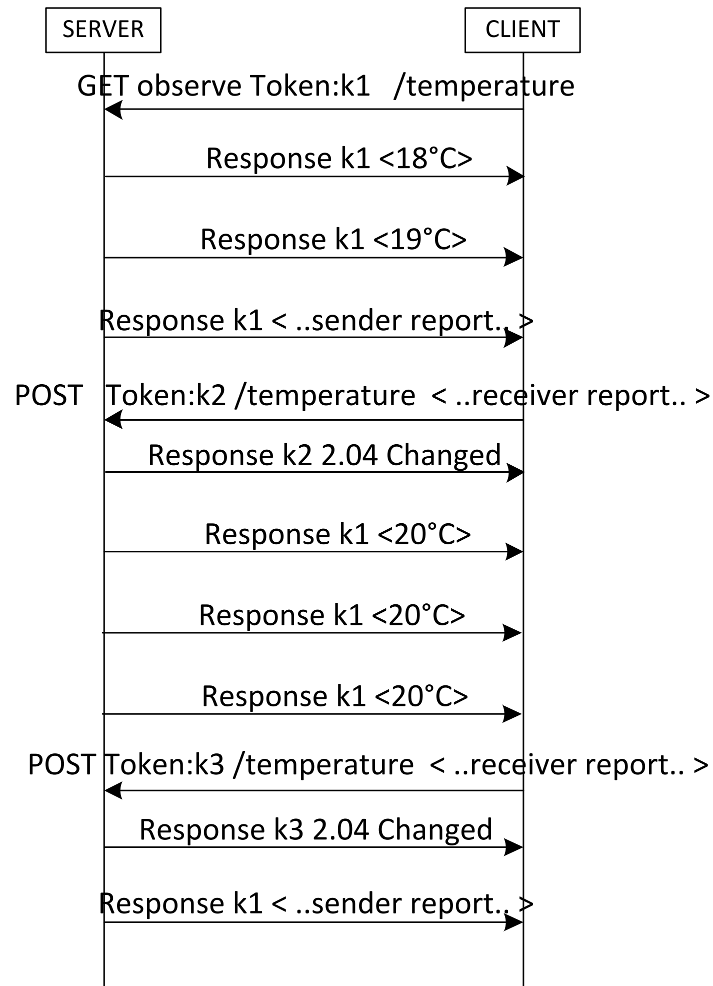

Figure 4.7: *CoAP exchange as implemented in this work.*

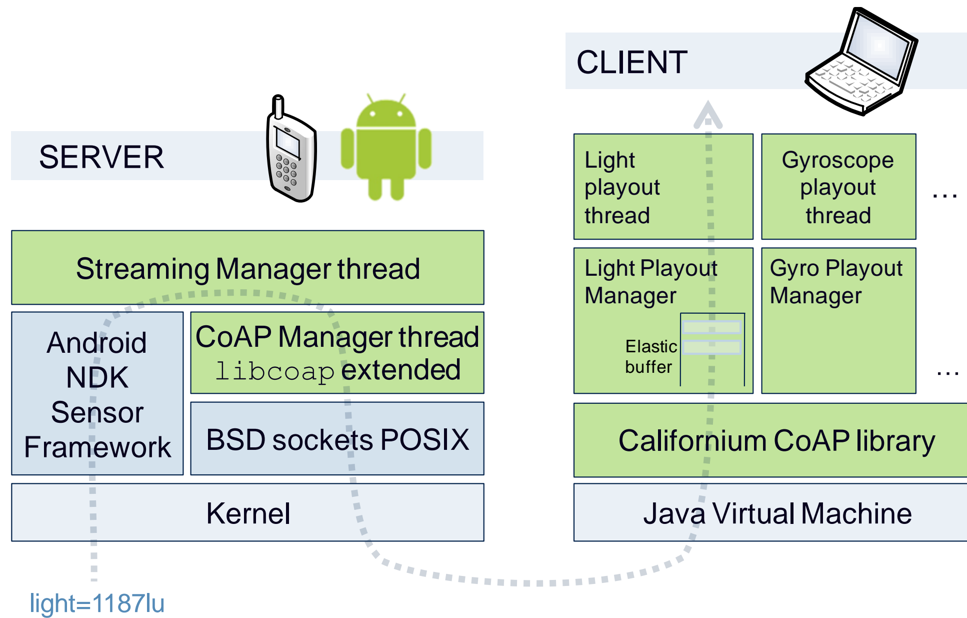

Figure 4.8: *View of modules that form the system.*

audio, video or sensor fusions created by the server. As an example:

`/temperature/critical?above=45`

may identify a temperature sensor which should notifiy interested observers when the temperature reaches 45 degrees.

The CoAP server runs on the Android device, is written in C and is based on `libcoap` [2] [16]. We extended `libcoap` to support CoAP Observe and to be robust against UDP packet duplication. The CoAP client is meant to run on a normal machine, is written in Java and is based on the Californium framework [14]. Server and client are both open-sourced [1] [2] [3]. A global view of the modules that form our system is given in Figure 4.8, where it is possible to follow the path of a sensor reading from capture to playout.

[1] `https://github.com/mzavatta/ZeSenseServer`
[2] `https://github.com/mzavatta/libcoap-zesense`
[3] `https://github.com/mzavatta/ZeSenseClient`

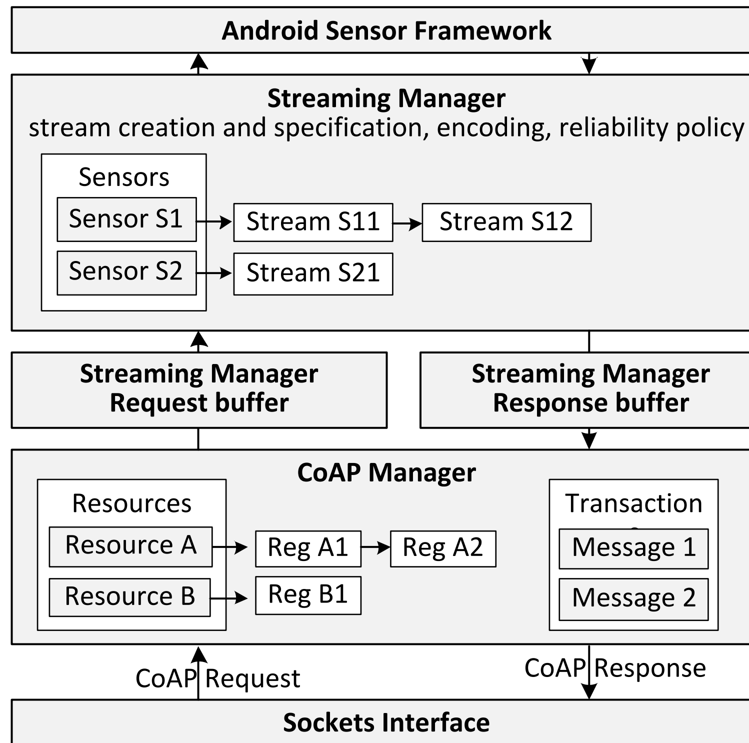

Figure 4.9: *View of modules that form the server. Reg blocks represent active Observe registrations.*

4.5 Server Implementation

The server architecture is composed of three main modules: the Android sensor framework (AS), the Streaming Manager (SM) and the CoAP Manager (CM). Each layer acts as an asynchronous service provider. The Android sensor framework is part of the Android platoform. It supplies sensor measurements to Android applications. The Streaming Manager is concerned with the creation, specification and packaging of sensor streams. It has been developed from scratch. The CoAP Manager is concerned with both levels of the CoAP protocol and acts as the server for the remote CoAP client. It is based on our extended version of `libcoap`. An overview of the server system is given in Figure 4.9.

The server is multi-threaded. Each module runs in its own thread. This was chosen to keep the CoAP protocol logic isolated from the interaction with

other potentially blocking interfaces. In this way CoAP transaction management can run concurrently to the retrieval of resource representations, which might be blocking and in streaming applications happens very frequently. Message passing was chosen to implement thread communication in order to minimize the sources of deadlocks.

4.5.1 Sensors in Android

The Android operating system is based on a Linux kernel. Applications in Android are usually written on the Java API and run within a modified version of the Java virtual machine called Dalvik. Processes and threads created in Java by user applications are mapped by Dalvik to their native Linux counterparts.

Linux implements many scheduling policies. Tasks usually start in the "normal" class. Normal-class tasks are scheduled by the Completely Fair Scheduler (CFS), used by Android as well. Within this class it is possible to escalate the task priority by modifying the *niceness* value, although the effectiveness of a higher niceness in practice is not a priori quantifiable. Mainstream Linux also has a Real-Time scheduling class, whose tasks get the highest priority. When ready, RT tasks can preempt others belonging to lower classes. It is not possible for Android applications to escalate a specific task to real-time priority, as this requires root privileges, although it is possible to change the niceness value. As a consequence, regular Android applications are not given temporal guarantees on execution. This influences the stability of the sampling rate of sensors.

Dalvik supports the Java Native Interface. This allows Java applications to call into code written and compiled in native language. Google releases a compilation toolchain and a set of native APIs called the Native Development Kit (NDK) to ease developers in writing native Android applications. The NDK API includes a version of the standard C library, OpenGL, a sensor library and many more. Native applications still run in a Dalvik instance but are not affected by the virtual machine management overhead. For example, an app thread that is running

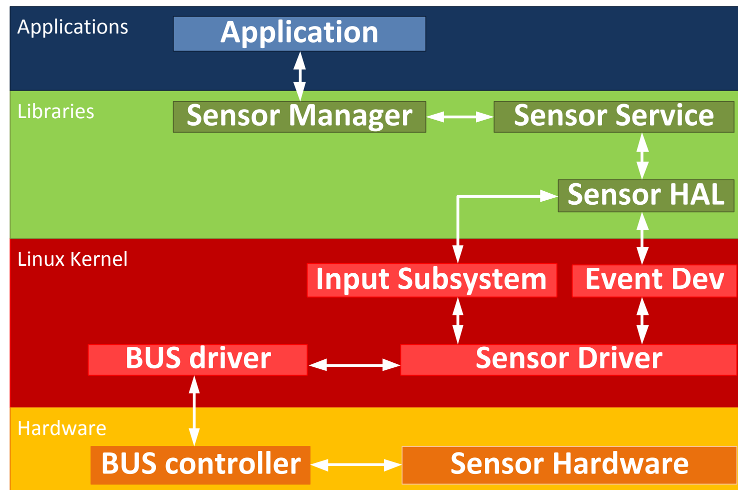

Figure 4.10: *The Android sensor stack [13].*

native code and that is not referencing any Java object does not need to stop when the garbage collector operates. Garbage collector activity is a major source of execution latency and jitter for interpreted languages like Java. Despite being less intuitive then Java development, native development is encouraged for real-time applications and facilities like OpenSL low-latency audio are available to developers. For these reasons we implemented our prototype with NDK.

Mobile devices are equipped with many sensors. For instance the Samsung GT-i9300 embeds the following sensors: accelerometer, gyroscope, proximity, light, pressure. The GPS can be considered a sensor as well. Sensor hardware is handled by Android according to Figure 4.10. The lowest level software module is the sensor driver. The driver is responsible for writing and reading signals and registers to control the hardware logic and fetch sensor measurements. When a measurement is ready, it is pushed by the driver into Linux's Input Subsystem. The Input Subsystem assigns a timestamp to each sample, which we call *OS timestamp* (OST), that corresponds to a reading of the system clock at the time of insertion. The Input Subsystem is the boundary between kernel-space and user-space. User-space applications can eventually fetch measurements and

their OST either directly via the Input Subsystem interface or via native or Java libraries.

Generic Sampling Model

Let us take a real example to dissect the sampling process. We look at the driver of the gyroscope inside Samsung Galaxy GT-i9300. The sensor is manufactured by ST Microelectronics, product code LSM330DLC. This example can be extended to most sensors as design principles of Analog-to-Digital conversion and I/O operations are similar.

Listing 4.1: *Excerpt of the LSM330DLC gyroscope driver implementation in Samsung Galaxy GT-i9300*

```
if (data->client->irq >= 0) { /* interrupt */
  ...
err = request_threaded_irq(data->client->irq, NULL,
      lsm330dlc_gyro_interrupt_thread\
      , IRQF_TRIGGER_RISING | IRQF_ONESHOT,\
        "lsm330dlc_gyro", data);
...
} else {
  ...
  hrtimer_init(&data->timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
  data->timer.function = lsm330dlc_gyro_timer_func;
  ...
  data->lsm330dlc_gyro_wq \
  = create_singlethread_workqueue("lsm330dlc_gyro_wq");
  ...
  INIT_WORK(&data->work, lsm330dlc_gyro_work_func);
}

static enum hrtimer_restart lsm330dlc_gyro_timer_func(struct
   hrtimer *timer)
{
  ...
  queue_work(gyro_data->lsm330dlc_gyro_wq, &gyro_data->work);
  hrtimer_forward_now(&gyro_data->timer, gyro_data->polling_delay
     );
  ...
}


static void lsm330dlc_gyro_work_func(struct work_struct *work)  {
  ...
  res = lsm330dlc_gyro_read_values(data->client,
```

```
    &data->xyz_data, 0);
  ...
  input_report_rel(data->input_dev, REL_RX, gyro_adjusted[0]);
  input_report_rel(data->input_dev, REL_RY, gyro_adjusted[1]);
  input_report_rel(data->input_dev, REL_RZ, gyro_adjusted[2]);
  input_sync(data->input_dev);
  ...
}


static irqreturn_t lsm330dlc_gyro_interrupt_thread(int irq\
  , void *lsm330dlc_gyro_data_p) {
  ...
  struct lsm330dlc_gyro_data *data = lsm330dlc_gyro_data_p;
  ...
  res = lsm330dlc_gyro_report_values(data);
  ...
}

static int lsm330dlc_gyro_report_values\
  (struct lsm330dlc_gyro_data *data) {
  ...
  res = lsm330dlc_gyro_read_values(data->client,
    &data->xyz_data, data->entries);
  ...
  input_report_rel(data->input_dev, REL_RX, gyro_adjusted[0]);
  input_report_rel(data->input_dev, REL_RY, gyro_adjusted[1]);
  input_report_rel(data->input_dev, REL_RZ, gyro_adjusted[2]);
  input_sync(data->input_dev);
  ...
}
```

Management of the sensor can be configured to be interrupt-based or polling based. In case of interrupts, the interrupt service routine (ISR) bottom half is implemented via threaded interrupt handlers. In case of polling, at each timer expiration (high resolution timer, *hrt*) the driver instantiates a workqueue.

Since workqueues and threaded IRQs run in kernel threads, the data read (DR) of the sensor registers happens inside fully schedulable tasks. These are subjected to CFS so the task scheduling latency is variable. At the lowest level, the data read corresponds to a reading of the sensor hardware internal register via an I2C bus. The rate and the policy with which the internal register is filled with samples varies across different manufacturers and drivers. The LSM330DLC itself allows many internal buffering and replacement policies. Furthermore, variable bus com-

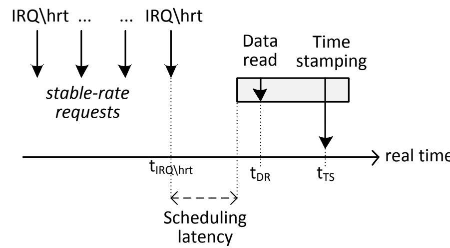

Figure 4.11: *Generic sensor sampling model. Intervals between $t_{IRQ\backslash hrt}$, t_{DR} and t_{OST} are unpredictable.*

munication delays may exist. Thus it is difficult to know when the measurement fetched by `lsm330dlc_gyro_read_values()` was actually performed. Even if the IRQ rate or the timer firing rate is perfectly stable, there can be variance on the DR rate.

When the data is read, it is pushed to the input subsystem and assigned an OST within the body of `input_sync()`. The execution might be preempted for whatever reason, for example by an ISR's top half. Thus the time between DR and OST assignment is variable, too.

In order to formally characterize the sampling mechanism, we define a model that abstracts these low level, platform-dependent details. We assume that IRQs or timer events do not exhibit any jitter. In case of interrupts, we assume that the top half simply acknowledges the IRQ and defers I/O to a threaded bottom half. We assume that the data read instruction instantaneously reads the physical quantity as seen by the sensor. Due to uncertainty in scheduling and preemptability of the ISR/workqueue, the interval between these actions is unknown, as illustrated in Figure 4.11. According to this model the sampling rate is randomly variable. Moreover, since there is a random delay between the sample's actual data read and the system clock reading, we cannot rely on OST to identify the data read instant, as shown in Figure 4.12.

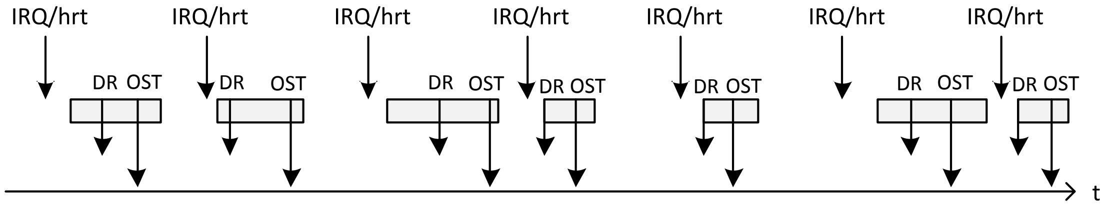

Figure 4.12: *Sampling random variability. Possible outcome of sampling and timestamping over time.*

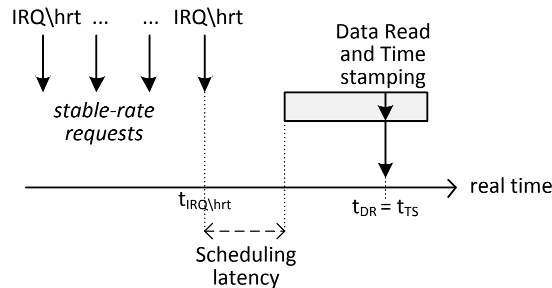

Figure 4.13: *Simplified sensor sampling model. Data read and timestamp assignment happen simultaneously.*

Simplified Sampling Model

Aware of making a simplification, we assume DR and OST to happen simultaneously (or, equivalently, in offset of a fixed period). The advantage of this model is that user-space applications become aware of data read times even though they can only observe OST. The disadvantage is that it introduces a synchronization error. The error can be reduced without changing the non-real-time nature of the operating system with sensor synchronization algorithms found in the literature, for example [20].

With this model in mind, we attempted to quantitatively characterize the sampling (data read) jitter on our device. That is, referring to Figure 2.7, the source-side portion outside the timestamp zone. We activate sampling of a sensor

at a certain rate and collect samples in user-space via NDK APIs. When a sample is collected we register the system time in a variable named collection timestamp (CLT). Each sample also carries the OST assigned by lower layers. We therefore have a sequence of pairs of timestamps:

$$(OST_1, CLT_1), (OST_2, CLT_2), (OST_3, CLT_3) \dots$$

We take average and standard deviation of three datasets:

Generation Periods

$OST_2 - OST_1$, $OST_3 - OST_2$, ...

Collection Periods

$CLT2 - CLT_1$, $CLT_3 - CLT_2$, ...

Travel Times

$CLT_1 - OST_1$, $CLT_2 - OST_2$, $CLT_3 - OST_3$, ...

Sampling is run for 30s. We measure activate the accelerometer and light sensors varying, one at a time: sampling rate, niceness of the collection task and overall system load. Results are shown in Figure 4.14.

The accelerometer data set shows two important features: for low sampling frequencies, the system is able to provide an actual sampling frequency very close to the requested one. For higher frequencies the sensor saturates. Indeed when we request 150 Hz, the actual arrival rate is close to 100 Hz. We believe that this correction is platform dependent. Also, as showed in the last row, a niceness value of -20 (highest priority) has been used for the user-space collection task. Beneficial effects on the speed and stability of the collection process are evident, even though the real benefits for user experience will likely be unnoticeable.

The light sensor data set shows the effect of system load. The load condition “camera preview” means that, during the 30s of sampling, in the display of the phone was shown the real-time video captured by the camera. We selected this

ACCELEROMETER

requested rate	Generation Rate	Generation Period avg	Generation Period sd	Generation Period sd / avg	Collection Period avg	Collection Period sd	Collection Period sd / avg	Travel Time avg	Travel Time sd	Travel Time sd / avg	niceness
Hz	Hz	msec	msec		msec	msec		msec	msec		default = 0
5	4.99	200.23	0.79	0.39%	200.30	1.73	0.865%	0.50	1.49	299.60%	0
10	9.99	100.13	0.30	0.30%	100.13	0.32	0.323%	0.12	0.11	90.52%	0
20	19.93	50.17	0.89	1.76%	50.17	1.06	2.105%	0.13	0.40	318.25%	0
50	49.42	20.23	1.31	6.46%	20.23	2.83	13.974%	0.52	2.22	409.76%	0
100	95.98	10.42	1.85	17.78%	10.42	4.84	46.302%	1.45	3.43	275.33%	0
150	95.93	10.42	1.82	17.50%	10.42	4.88	46.838%	1.41	3.89	281.98%	0
150	97.75	10.23	1.20	11.69%	10.23	1.62	15.855%	0.17	1.02	605.36%	-20

LIGHT

requested rate	Generation Rate	Generation Period avg	Generation Period sd	Generation Period sd / avg	Collection Period avg	Collection Period sd	Collection Period sd / avg	Travel Time avg	Travel Time sd	Travel Time sd / avg	load
Hz	Hz	msec	msec		msec	msec		msec	msec		default = 0
20	20.00	50.00	0.67	1.34%	49.96	1.34	2.69%	0.12	1.18	869.22%	free
20	20.00	50.00	0.39	0.78%	49.93	1.71	3.42%	0.19	1.67	857.93%	camera preview
50	50.00	20.00	0.30	1.49%	20.00	0.44	2.22%	0.07	0.24	300.59%	free
50	50.00	20.00	2.67	13.34%	19.99	2.69	13.45%	0.12	0.30	260.48%	camera preview
100	100.00	10.00	0.26	2.60%	10.00	0.31	3.12%	0.05	0.13	267.44%	free
100	99.83	10.02	2.82	28.12%	10.01	2.84	28.37%	0.11	0.20	180.39%	camera preview
150	166.65	6.00	0.16	2.72%	6.00	0.21	3.47%	0.04	0.10	237.24%	free
150	164.89	6.07	1.98	32.51%	6.07	2.01	33.09%	0.10	0.25	243.28%	camera preview
300	331.68	3.01	0.40	13.37%	3.01	0.49	16.29%	0.05	0.24	470.05%	free
300	315.82	3.17	1.39	44.01%	3.17	1.43	45.06%	0.08	0.21	262.42%	camera preview
600	743.36	1.35	0.10	7.27%	1.35	0.36	26.57%	0.09	0.56	635.33%	free
600	769.30	1.30	0.43	33.44%	1.30	0.64	49.40%	0.15	0.61	401.85%	camera preview

Figure 4.14: *Sampling results.*

load condition because it follows our use case and it is, as well as sampling, another I/O-intensive operation. For every requested frequency, performance of sampling decreases with respect to the unloaded case. Finally, also for the light sensor the system is not able to provide exactly the requested rate.

The Android documentation suggests 50 Hz as a sampling rate suitable for gaming applications. We consider rates $\leq$ 50 Hz as appropriate for our use case. The generation period coefficient of variation does not exceed 13% in the worst case, which corresponds to 2.6 ms over a nominal period of 20 ms. Also, in both datasets, the coefficient of variation improves as the frequency becomes lower. The discrepancy between nominal and real sampling period is a source-side component that adds to end-to-end presentation jitter and skew. It was mentioned in Section 2.1.2 that upper bounds for presentation jitter and skew are 10 ms and 160 ms respectively. Although these studies target audiovisual experience and not presentation of scalar values, they are the most suitable thresholds that have been found in the literature. Facing 10 ms jitter and 160 ms skew upper bounds, 2.6 ms worst-case standard deviation is considered acceptable. We postpone a final assessment eventually on the visual performance of our prototype. Note that such results would not provide meaningful insights within the generic sampling model because generation periods would not actually correspond to data read periods. No conclusion on the sampling performance could be drawn in such a case.

Most of the sensors in our Samsung GT-i9300 are sampling based. The application specifies a desired sampling rate, the Android system instructs the sensor and tries to push data at the desired rate. However the proximity sensor is event based. It is activated by the same commands as the other sensors in the platform, although samples become available to the application only upon relevant changes in the observed physical phenomenon, regardless of the requested rate. We are unaware if the driver is sampling based and operates some kind of filtering or it is really the hardware that behaves in this fashion. This random event behavior

	Class	Axes	Max Frequency
Accelerometer	sampling based	3	100 Hz
Gyroscope	sampling based	3	200 Hz
Light	sampling based	1	over 600 Hz
Proximity	event based	1	-

Table 4.2: *Sensors used in this work and their main characteristics.*

is, by the way, also a well-known characteristic of the GPS "sensor". Thus the proximity sensor creates an unknown-frequency stream.

Android Sensor Framework Interface

The main operations offered by the Android NDK sensor API are:

- Enable and disable a sensor
- Change sensor frequency
- Create and destroy an event queue
- Fetch samples from the event queue

It is possible to specify an initial sampling frequency for the sensor and later change it on-the-fly. For event-based sensors, the sampling frequency parameter is ignored. The service is asynchronous. Clients, such as our Streaming Manager, fetch samples from a FIFO event queue. The data structure that represents the sample mainly contains the physical quantity measurement and its OST. Since there is no indication of the sampling frequency in the data structure, if the sampling frequency is changed on-the-fly, it is not trivial to recognize the first sample generated with the new sampling frequency. It can only be done by inspecting the difference between the OST of two consecutive samples.

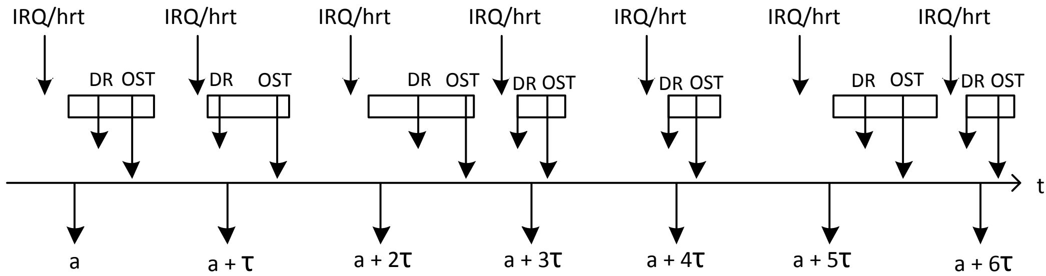

Figure 4.15: *Timestamp assignment in this work.*

4.5.2 Timestamp Assignment

As by RTP guidelines, timestamps should be assigned from a fictitious timestamp clock by incrementing an initial random timestamp of a fixed amount, knowing the timestamp clock frequency f_{TS} and the sampling frequency λ. Each stream should have its own, independent timestamp clock. We employ this technique in assigning TSs that are transmitted to the sink for intra-stream synchronization.

However, OSTs are still useful for inter-stream synchronization, as they provide the most accurate sample position on a time reference uniform across different streams. Therefore, the notion of wall clock is naturally implemented by the source host system clock. Note that, regardless of the timestamp convention used, both intra-stream synchronization and inter-stream synchronization will suffer an error due to the DR rate uncertainty explained above.

4.5.3 Streaming Manager

The Streaming Manager is a server of sensor streams. It produces well-defined and specified streams starting from single measurements.

The first task of SM is to collaborate with AS in order to collect measurements that make up a stream, so SM uses the AS interface to enable sensors and collect samples. As described previously, it is desirable to have known-frequency sensor streams. Issues arise when dealing with event-based sensors such as proximity.

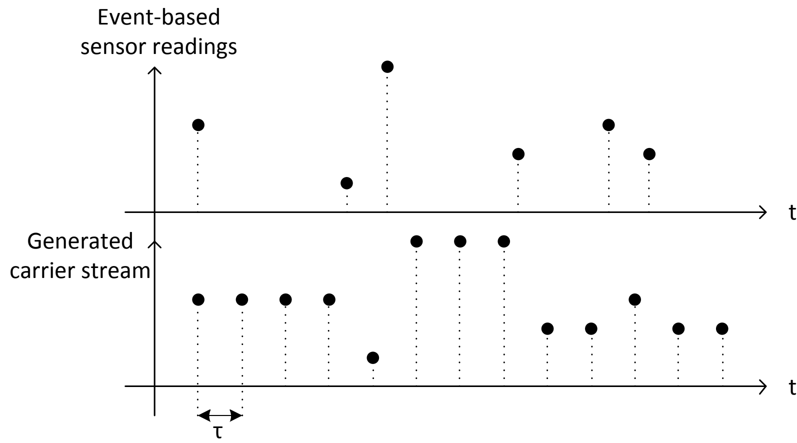

Figure 4.16: *Carrier stream creation.*

For this class the Streaming Manager creates a carrier stream. When SM is asked to stream an event-based sensor, it performs two actions: it enables the sensor in AS and it spawns a new thread. Measurements are not directly sent to the client but are stored in a one-slot cache that holds the last-known measurement. The thread issues fake samples at a constant rate. Each fake sample bears the last known measurement, which is found in the cache. The process is outlined in Figure 4.16. This method is simple and effective but it has some disadvantages. First, an event might need to wait some time, at most τ, before being reflected in a carrier sample. This causes a slight loss of synchronization (outside the timestamp zone). Also, if a newer measurement replaces an old one soon after, no carrier sample might be able to transport the old one. Events are guaranteed to be reflected on the carrier if they are at most as frequent as λ. The choice of the carrier frequency should take into account the characteristics of the phenomenon that is being measured.

The Streaming Manager is also in charge of generating the stream specification. For every sample, it generates a timestamp TS to be used for intra-stream synchronization, using the mechanism described in previous sections. It also maps TS to a wall clock instant. Since all OSTs refer to the same time reference, the

choice of OSTs for inter-stream synchronization becomes natural. Their time source is the internal Android system clock, given by:

```
clock_gettime(CLOCK_MONOTONIC, &t);
```

We take this clock source as the wall clock. This clock is monotonic across a system uptime and does not depend on external factors such as NTP or GPS clock updates, processor sleep or standby sessions, so it perfectly fits our needs. It runs at 1 GHz. The clock mapping between timestamps and wall clock finally becomes $(TS, f_{TS}, OST, f_{WC})$ where $f_{WC} = 1GHz$.

The Streaming Manager serves also as samples encoder. In principle a network packet could contain many samples from different sensors. Various patterns are adopted to packetize samples, but it is intentionally avoided to mix samples from different streams in the same packet. Against potential bandwidth savings is the fact that mixing different streams in one packet multiplies the effect of packet loss or excessive delay. On the other hand we do aggregate in one packet samples belonging to the same stream.

Each stream might use its own reliability policy based on the intrinsic characteristics of the media or the client requests. The SM controls reliability on a per-stream basis. It takes care of forward error correction directly by buffering old samples and repeating them in future packets. It instructs CS to use CoAP Confirmable Messages for backward error correction.

SM's key data entity is the stream. The structure is shown in Listing 4.2. The `event_buffer` aids in sample packaging. `retransmit` and `repeat` flags define the reliability policy. The stream specification is derived from the stream frequency and the timing records.

Listing 4.2: *Data structure that represents a stream in Streaming Manager*

```
typedef struct ze_stream_t {
  struct ze_stream_t *next;
```

```
    /* Stream ticket. */
    ticket_t reg;

    /* Event buffer. */
    ASensorEvent event_buffer[SOURCE_BUFFER_SIZE];
    int event_rtpts_buffer[SOURCE_BUFFER_SIZE];
    int event_buffer_level;

    /* Reliability policy. */
    int retransmit;
    int repeat;

    /* Stream frequency. */
    int freq;

    /* Timing records. */
    uint64_t last_wts;  //Last wallclock timestamp
    int last_rtpts; //Last timestamp

} ze_stream_t;
```

The Streaming Manager also keeps a register of the sensors available on the Android platform. Important sensor attributes are the current sampling frequency, a cache to quickly retrieve the last-know measurement and a reference to the carrier thread interface, if applicable. More than one stream can be associated to each sensor. In fact many observers can register for updates from the same sensor and each might require different streaming parameters e.g. sampling rate, packetization, reliability etc.

The Streaming Manager service invocation interface follows a well-known pattern called Asynchronous Completion Token [24]. It resembles a classic request/response protocol, not different from the concepts used in CoAP itself. This pattern allows a client to request asynchronous services. Our version is designed to be used on lossless, FIFO communication channels.

In our implementation of the pattern, SM Requests are of three types: START STREAM, STOP STREAM, ONESHOT. START STREAM asks the server to start a new sensor stream. ONESHOT simply asks for one sensor measurement. A stream stays active indefinitely until a STOP STREAM request arrives. Every request carries a *ticket*, and every response relative to that request will carry the

same ticket. The ticket is useful to clients to demultiplex the response to the proper handler.

SM Responses are of type STREAM UPDATE, ONESHOT and STREAM STOPPED. STREAM UPDATE responses carry sensor samples and stream metadata. ONESHOT responses simply carry one sensor sample. The STREAM STOPPED response is issued after a STOP STREAM request to guarantee ticket expiration i.e. that it will no longer appear in responses. The SM Response data structure shown in Listing 4.3.

Listing 4.3: *Data structure that represents a Streaming Manager response*

```
typedef struct ze_sm_response_t {

  /* Response header. */
  int rtype;     // Response type
  ticket_t ticket;  // Response ticket

  /* Application-specific payload. */
  unsigned char *pk;

} ze_sm_response_t;
```

It is made of a header and a payload. The header is independent of the application and only includes the response type and the ticket. `pk` points to the application-specific response payload. When `rtype` is STREAM STOPPED, there is no payload. When `rtype` is either STREAM UPDATE or ONESHOT, `pk` has the structure outlined in Listing 4.4.

Listing 4.4: *Data structure that represents a Streaming Manager response payload*

```
typedef struct {

  /* Timing parameters. */
  int64_t ntpts;
  int rtpts;

  /* Reliability policy. */
  int conf;

  /* CoAP payload formatted according to our CoAP extension. */
```

```
  unsigned char *data;
  int length;
} ze_sm_packet_t;
```

`*data` contains the CoAP payload, formatted according to our CoAP extension (Figure 4.5), that CM should send to CoAP clients. `conf` dictates the reliability policy that CM should use. Timing parameters are useful for CM to create Sender Reports.

Due to this well-defined and decoupled interface, the SM is a general purpose module. It can be used with any other underlying transport protocol, not necessarily our CoAP implementation.

To summarize, SM listens for requests from CM, in turn using Android as the server for bare sensor measurements. Whenever a measurement is ready, the SM maps it to a stream. When a packet is complete, it is passed to CM for remote delivery.

4.5.4 CoAP Manager

The CoAP Manager mainly performs the algorithms of the CoAP protocol. It shares its processor time between listening to CoAP messages, acting on CoAP requests, sending responses and retransmitting failed Confirmable messages. CoAP requests are dispatched according to a resource register. Retransmissions and message duplication is controlled according to a transaction register.

It relies on the Streaming Manager to provide resource representations i.e. sensor measurements. The service invocation is asynchronous so that CM can still serve CoAP requests while SM is working. There is a close correspondence between a CoAP Observe registration and a stream as intended by the Streaming Manager, but the concepts are totally decoupled. A possible CM-SM exchange is given in Figure 4.17. It is evident the asynchronous nature of the communication. Messages are stored in buffers and their collection depends on the read frequency of the receiving thread.

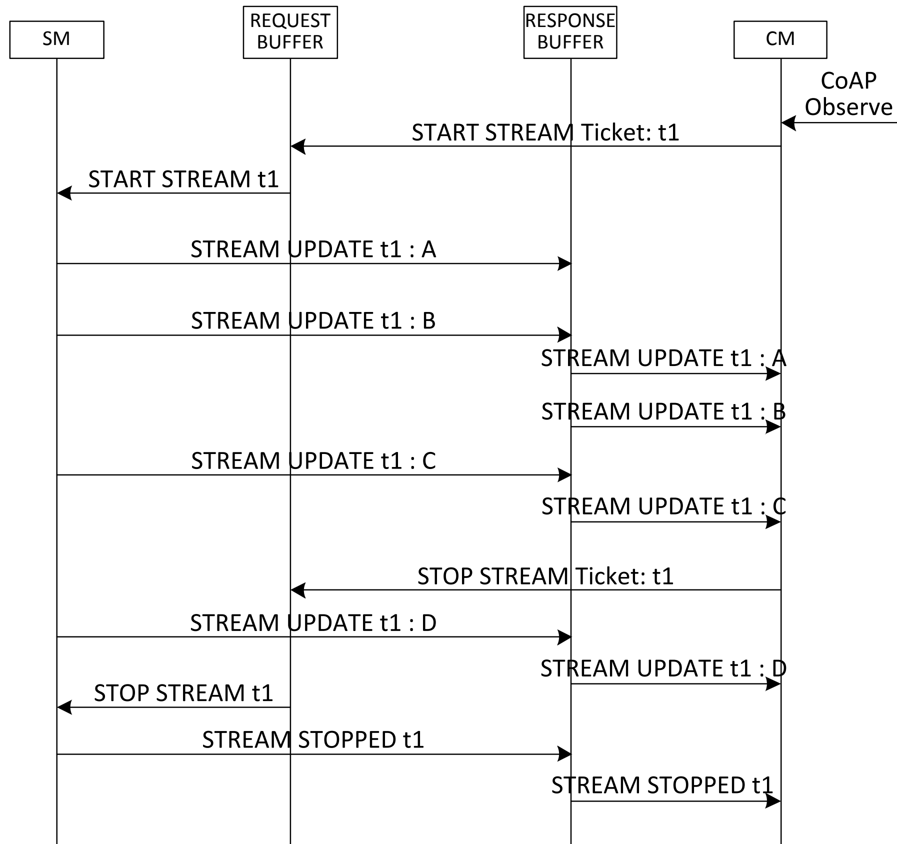

Figure 4.17: *Possible outcome of the Streaming Manager service invocation by CoAP Manager.*

In our implementation the ticket takes the form of the memory address of the Observe registration record. In this way, every time the CM gets a response from SM, it can dereference the ticket received in the response and get instant access to all the state information regarding the Observe registration. This avoids searching the record in the registration register every time a SM response arrives, which in a streaming application can happen hundreds of times per second. Thus demultiplexing overhead is very limited. Using a reference counter we make sure that the Observation record memory area is not deallocated or reallocated before a STREAM STOPPED response has arrived. This is consistent because the protocol guarantees no other response bearing the ticket after a STREAM STOPPED response has been received, and because our buffers ensure no message loss.

4.5.5 Message Buffers

The message buffers are the communication channel between SM and CM. They address a producer/consumer problem. CM and SM share two buffers: the request buffer and the response buffer. CM is the producer of the request buffer while SM is the producer of the response buffer. Producer-consumer synchronization is required because threads do not operate exactly at the same speed [23]. Since buffers are shared memory it is important to use synchronization primitives to ensure state consistency.

Our buffers are bounded and work in a FIFO manner. They are not circular, meaning that the oldest item is not overwritten if the buffer is full. Two functions are provided, put() and get() that are able to write and read one item. `put_request()` implements a blocking write and `get_request()` a non-blocking read. `put_response()` implements a timed blocking write and `get_response()` a non-blocking read. POSIX mutexes and condition variables implement mutual exclusion and blocking.

Listing 4.5: *get() and put() implementation of the response buffer. put blocks for a maximum time. get is non-blocking.*

```
ze_sm_response_t get_response_buf_item(ze_sm_response_buf_t *buf)
    {

  ze_sm_response_t temp;

  /* Synchronize with producer. */
  pthread_mutex_lock(&(buf->mtx));
    if (buf->counter <= 0) {
      /* Buffer empty, return invalid item. */
      temp.rtype = INVALID_COMMAND;
    }
    else {
      /* Copy item from buffer head. */
      temp = buf->rbuf[buf->gethere];

      /* Advance buffer head. */
      buf->gethere = ((buf->gethere)+1) % COAP_RBUF_SIZE;

      /* Decrease item count. */
```

```
      buf->counter--;

      /* No longer full. */
      pthread_cond_signal(&(buf->notfull));
    }
  pthread_mutex_unlock(&(buf->mtx));

  return temp;
}
int put_response_buf_item(ze_sm_response_buf_t *buf, int rtype,
    ticket_t ticket, int conf, unsigned char *pk) {

  int timeout = 0;
  struct timespec abstimeout;

  /* Synchronize with consumer. */
  pthread_mutex_lock(&(buf->mtx));
    if (buf->counter >= COAP_RBUF_SIZE) {
      /* Buffer full, wait for some time. */
      clock_gettime(CLOCK_REALTIME, &abstimeout);
      abstimeout.tv_sec = abstimeout.tv_sec + 2;
      timeout = pthread_cond_timedwait(&(buf->notfull), &(buf->
         mtx), &abstimeout);
    }

    if ( !timeout ) {
      /* Insert item in buffer tail. */
      buf->rbuf[buf->puthere].rtype = rtype;
      buf->rbuf[buf->puthere].ticket = ticket;
      buf->rbuf[buf->puthere].pk = pk;

      /* Advance buffer tail. */
      buf->puthere = ((buf->puthere)+1) % COAP_RBUF_SIZE;

      /* Increase item count. */
      buf->counter++;
    }
  pthread_mutex_unlock(&(buf->mtx));

  return timeout;
}
```

The timeout in `put_response()` prevents the occurrence of a deadlock. It would occur when both buffers are full and both threads attempt a write operation. When `put_response()` times out, SM reads some items from the request buffer before retrying the put. Reading unblocks CM and thus resolves the deadlock.

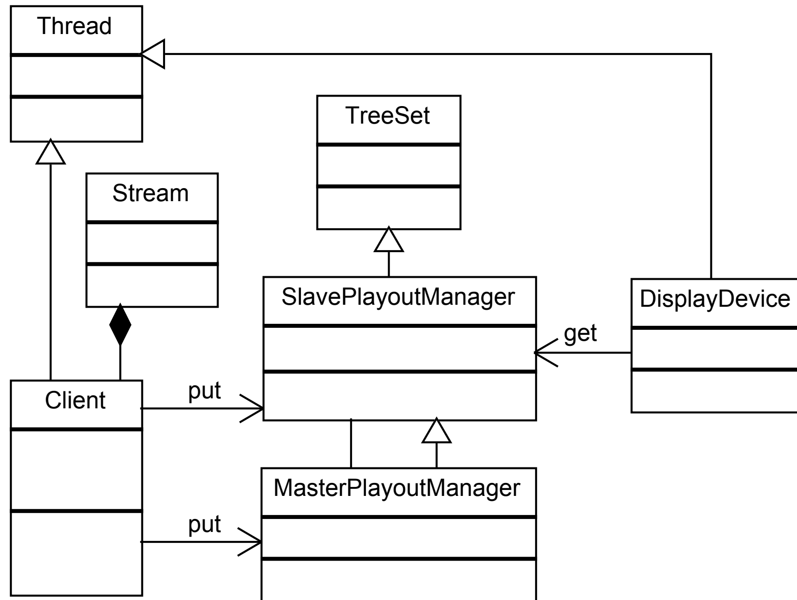

Figure 4.18: *Client class diagram.*

4.6 Client Implementation

The client is written in Java. It is based on the Californium CoAP library. The task of the client is to play sensor streams. A simplified class diagram is given in Figure 4.18.

We assume to have nonmalleable playout devices [6]. A nonmalleable device consumes data at a constant rate, which we call f_{PRQ}, typically issuing an interrupt request to the processor. We assume the playout interrupt rate to be an integer multiple of the sensor sampling rate to avoid complex resampling issues. This assumption is realistic as both Android sensor sampling and PC-monitor display are flexible with respect to frequency. Each playout device is simulated by a Java `Thread` (`DisplayDevice`) that calls a method at a constant rate.

For each sensor stream that the client wishes to play:

- A CoAP Observe request is sent
- An object of classes `Stream` and `SlavePlayoutManager` is instantiated

- A `DisplayDevice` is started
- Incoming CoAP samples are first mapped in `Stream` then inserted in `SlavePlayoutManager`

The class `Stream` represents a slave stream. It holds information such as the clock mapping timestamp-wall clock. The client runs the synchronization algorithm described in Section 4.2. The elastic buffer is implemented by the class `SlavePlayoutManager`. It extends the Java class `TreeSet` that implements an ordered set of elements. `SlavePlayoutManager`'s `get()` method is called at f_{PRQ} by the `DisplayDevice` and it runs Algorithm 2 at every call. The class `MasterPlayoutManager` is a singleton that represents the master stream state. The Master Playout Offset is an attribute of this class. Every instance of `SlavePlayoutManager` is connected to `MasterPlayoutManager` because the `get()` method uses MPO at every call.

MPO is set by the first sample of any stream that arrives to the client. The master stream is fictitious, it is simply summarized to the external world by MPO. In order to simulate presentation pauses or skips it is enough to vary MPO and all slaves will follow. The clock mapping is conveyed by CoAP sender reports.

Chapter 5

Experimental Evaluation

Metrics of the evaluation are those described in Section 4.1. SYNC is kept optimal by the synchronization algorithm, so the Playout Reliability coincides with the Delay Constrained Reliability. Note that PR (DCR) denominator refers to original samples. Also, DCR numerator counts unique samples. Since the layout and field length of the payload format is not optimal, network load is significant as a comparison metric and not as absolute value.

5.1 Experiment Variables

Different delivery strategies are compared for different values of packet loss rate:

Default

In the default scenario each Data packet carries one sample. They are sent in Non-confirmable messages. As advised by the standard, one every five Data packets is sent as Confirmable in order to confirm the client interest. Sender and Receiver Reports are Confirmable messages. This is the simplest strategy. It has the minimal features of a streaming service, similar to a simple use of RTP. It is possible to reduce the number of packets by aggregating more samples into one packet. The number of active CoAP transactions at any time is modest. Since error correction is absent, losing a packet means losing an entire information unit with no chance of recovery.

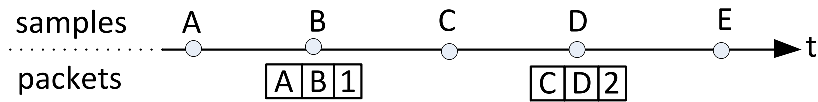

Figure 5.1: *Aggregation strategy with AF=2.*

Bandwidth-saving

In this strategy each packet contains more than one sample. We call Aggregation Factor (AF) the number of samples per packet. Figure 5.1 shows packetization for an AF=2. A buffer of $AF - 1$ cells is needed to store samples. This strategy allows to reduce header overhead. The impact is quite significant. Limiting the analysis to the UDP layer, for AF=2 and a one-axis sensor, savings are as much as 27%. In fact [1]:

- Two separate packets:
  (8 byte UDP header + 16 byte CoAP header + 4 byte Payload Header + 4 byte TS + 20 byte data) * 2 packets = 104 bytes
- One packet: 8 byte UDP header + 16 byte CoAP header + 4 byte Payload Header + (4 byte TS + 20 byte data) * 2 samples = 76 bytes

The drawback of aggregation is that the effect of packet loss and jitter is going to affect more than one sample at a time.

Reliability

To improve PR we can leverage backward error correction and forward error correction (or both simultaneously):

- *Retransmission*: every Data, Sender and Receiver Report is a Confirmable message. Data packets contain one sample. The number of

[1] The CoAP header figure accounts for 4 bytes fixed header, 8 bytes Token option, 2 bytes Observe option, 2 bytes Content-format option. It neglects option deltas. 20 bytes data accounts for an ASCII-encoded floating-point number.

packets flowing is at least twice the number of issued samples, because each message triggers at least an ack or a retransmission.

- *Repetition*: every packet contains two samples: the current one and a copy of a number of previous samples. The number is called Repetition Factor (RF). The case with RF=1 is shown in Figure 2.9. Messaging mimics the default strategy.

5.2 Parameter Setting

The device under test is the Samsung Galaxy GT-i9300 with Android 4.1.2. Three sensors are enabled: gyroscope, light and proximity. For example, the full URI for the proximity resource becomes:

```
coap://192.168.43.1:5683/proximity
```

Other settings:

- sampling frequency $\lambda = 10$ Hz
- retransmission timeout random within [ack_timeout, ack_timeout * 1,33] (as suggested by CoAP standard)
- playout request frequency $f_{PRQ} = 20$ Hz

The client runs on a virtualized Linux machine. The physical link is established via Wi-fi (Android device in hotspot mode). By running ping trials it can be seen that client-server RTT ≈ 6 ms. The Internet is simulated using `netem` (iproute2-ss130716). `netem` is a simulator common in Linux distributions that is able to recreate WAN characteristics: packet loss, network delay and jitter. We use the command:

```
netem delay 110ms 80ms distribution pareto loss x%
```

to simulate a pareto-distributed one-way delay and a packet loss of x%. However, the meaning of the delay values is not well documented. The packet loss figure is instead reliable. The above command applied to incoming and outgoing packets gives approximately the following results, observed using ping (thus include also the Wi-fi delay):

- $d_N \approx 75$ ms, RTT ≈ 150 ms
- d_N peaks of ≈ 50 ms to ≈ 400 ms, pareto distributed
- identical characteristics in both directions

The pareto distribution is suggested by [7]. 75ms as network delay is suggested by experiments over LTE networks conducted in [19]. The QoE parameters become:

- maximum end-to-end delay $d_{EE} = 300$ ms
- buffering delay $d_b = 225$ ms

In order to comply to the assumption of the synchronization algorithm, we make sure that the first packet sent by the server gets exactly d_N by using an on-purpose `netem` policy for it: `netem delay 75ms`

The service executes for 2 minutes, which gives about 3600 samples sent. Regarding delivery strategies, settings are AF=2 and RF=1.

5.3 Results

For clarity, some terminology is given:

- *duplicate*: refers to a sample copy that is sent after the original one. Retransmission generates a duplicate for each retransmitted message whereas repetition generates a duplicate for every sample.
- *useful duplicate*: duplicates received before deadline and before the original

- *false useful duplicate*: useful duplicates whose original arrived before deadline
- *true useful duplicate*: useful duplicates whose original was lost or late
- *duplicate utilization*:

$$\mathrm{DU} = \frac{\text{true useful duplicates}}{\text{total duplicates sent}}$$

The retransmission strategy impact is a function of the retransmission timeout (ack_timeout). The first experiment was aimed at finding its optimal value. The results are shown in Figure 5.2. DCR improves linearly as the timeout decreases. This is expected since duplicates sent earlier have more chances to arrive at destination before deadline. On the other hand, the amount of data exchanged by the server increases. Thus, there is the need of finding a tradeoff between quality and resource usage. The criteria used to choose the optimal ack_timeout is the duplicate utilization defined above. It measures the success rate of the mechanism, that is the fraction of duplicates that actually recover an original sample. The higher DU, the lower is the waste of retransmissions. As shown in Figure 5.2, utilization hits a maximum at 150 ms, which is close to the network round-trip time. Below 150 ms more retransmissions are useless because acks are not given enough time to arrive. Above 150 ms, more retransmissions are useless because they are sent too late and miss the delivery deadline. Note that this heuristic is application-specific, as it indeed depends on the deadline setting and it considers all samples as equally important.

Figure 5.3 shows the metrics varying delivery strategy and packet loss. The default strategy reaches a Delay Constrained Reliability of about 90% at 10% loss. It may seem that jitter has no negative effect, as the slack from perfect DCR is exactly as much as packet loss. However, some samples do miss the deadline by taking more than 300 ms network delay. A possible explanation is that a similar

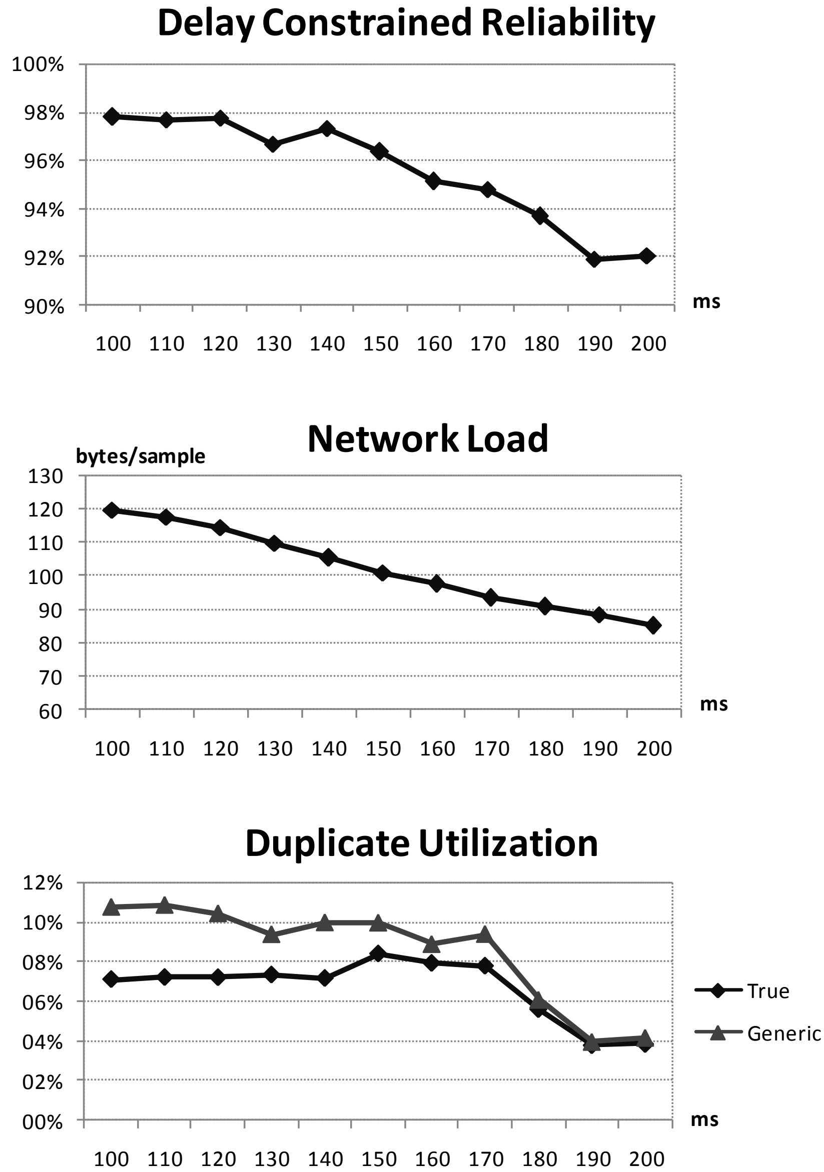

Figure 5.2: *Performance and load metrics for the retransmission strategy varying the retransmission timeout (ack_timeout). The network behaves according to* `netem delay 110ms 80ms distribution pareto loss 10%`. True *duplicate utilization refers to the metric as defined previously.* Generic *duplicate utilization refers to both true and false useful duplicates.*

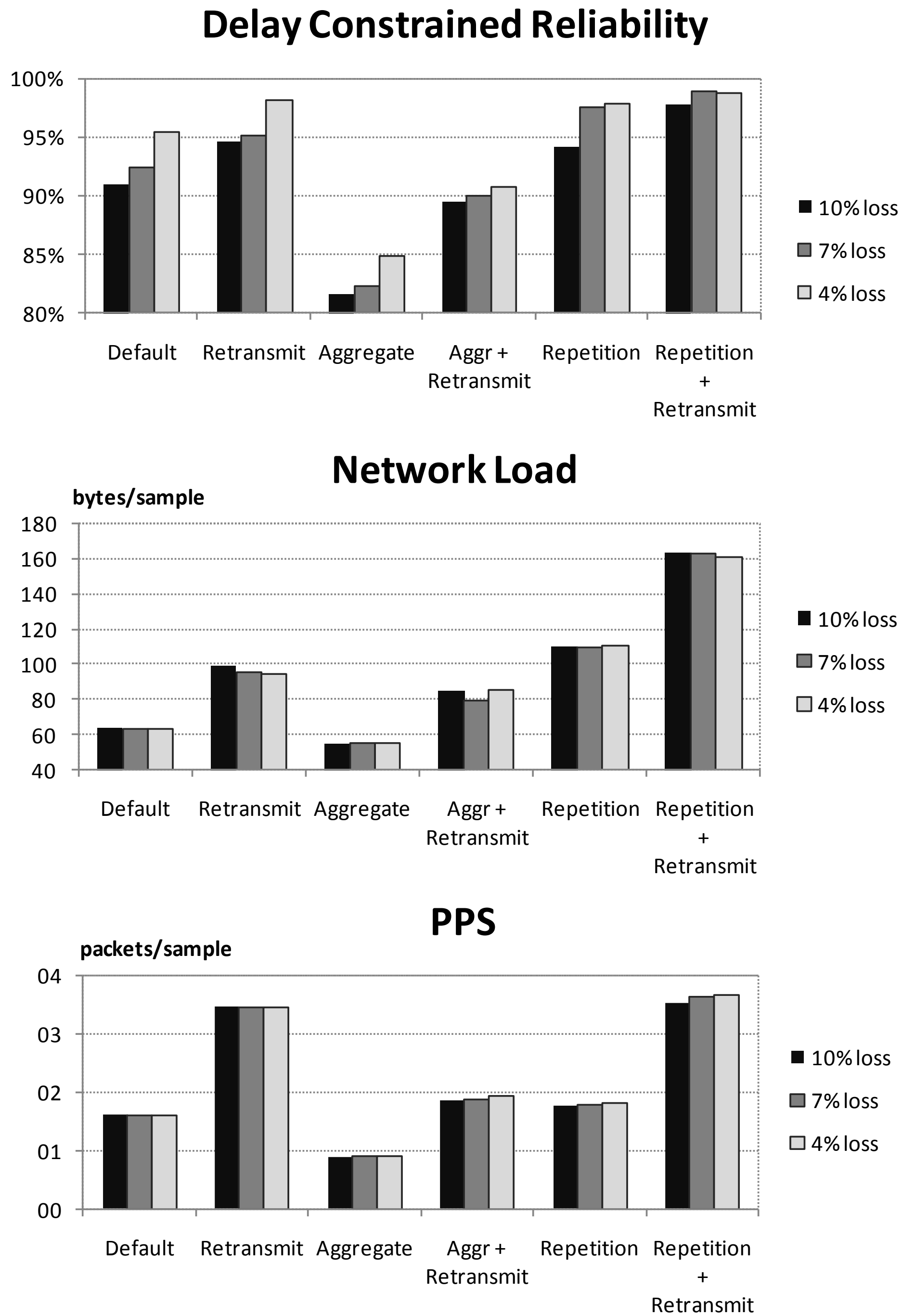

Figure 5.3: *Performance and load metrics for different delivery strategies and packet loss.*

number of samples is recovered by Confirmable messages, which are sent once every five Non-confirmables.

As expected the aggregation strategy reduces DCR considerably because a lost or late packet impacts on two samples rather than one. On the other hand, network load is not reduced significantly because savings amount only to the fixed header portions. It is interesting to notice the similarity between the default strategy and the aggregation strategy with retransmissions, both in DCR and in network traffic. Clearly aggregation with retransmission is more complex and computationally heavier.

Forward error correction and retransmission have similar benefits i.e. they recover the same amount of samples. Also the amount of redundancy needed for this achievement is similar (as indicated by the network load), so we might argue that the two techniques are equivalent. However, according to expectations, retransmission uses many more packets than repetition to send the same amount of data. Thus packets used by retransmission are smaller but more frequent. From a power consumption perspective, this may or may not be a desirable behavior depending on the underlying medium access mechanisms e.g. contention-based or scheduled. Note that retransmission and repetition can be used together to achieve even higher DCR at the expense of a considerable network load.

Clearly as the network becomes more and more reliable, the effects of error recovery techniques become less evident. The activation of error correction is able to increase $\approx 4.5\%$ DCR with respect to the default case at 10% loss. At 4% loss, the positive effect decreases to 2.5%. As well, the benefit of retransmission on aggregation grows from 5% at 4% loss to 9% at 10% loss.

It is interesting to note the behavior of the retransmission strategy varying the packet loss. As the error probability grows, one would expect the packet flow to increase because retransmissions happen more frequently. On the contrary, in Figure 5.3, there is no evidence of such behavior. As illustrated in Figure 5.4,

packets/sample actually decrease. The behavior is correct. Recall that the packet count includes incoming and outgoing UDP packets at the server interface. As packet loss increases, the outgoing UDP packets in the form of retransmissions increase. However the incoming UDP packets, carrying ACKs, decrease due to higher packet loss. As a confirmation, network load as observed by the server increases, since retransmission messages carry more data than acks. The discrepancy in Figure 5.4 from the expected 2 packets/sample [2] in perfect network condition is caused by two factors: Sender/Receiver Reports and retransmissions. Sender and Receiver Reports are included in the packet count but do not carry any sample. A small fraction of retransmissions occur even in this case because of a modeling simplification. ack_timeout is set exactly on the `netem` policy round-trip time (150 ms), assuming zero the Wi-fi delay jitter. When some packets (Data or the corresponding ack) get a relevant Wi-fi delay, the exchange misses ack_timeout and a retransmission is triggered. Also, the discrepancy in Figure 5.4 from DCR=1 in perfect network condition [3] is due to loss at the client (in fact in the experiment all samples do arrive at the sink before deadline). Loss in the client happens because the playout request rate is not perfectly stable, thus some playout times are not correctly requested (refer to Algorithm 2).

Finally, a word on visual Quality of Experience. In the prototype we display sensor readings on dial meters in a console on a laptop monitor. When a sample is displayed on time, the needle is colored for the whole sample period. When a sample is missing, the needle is turned light grey for one period. As an example, consider a smartphone moving from inside a room to the outside in a person's hand. The visual quality of experience depends on many factors: high sampling frequency and high DCR give a feeling of stream continuity. In this setting, 10 Hz offers an acceptably continuous view of the person movement in terms of changing light conditions, acceleration and deceleration of walking and corner turns. The

[2] One Data packet whose ack always returns by RTT.

[3] All samples arrive at the sink before deadline.

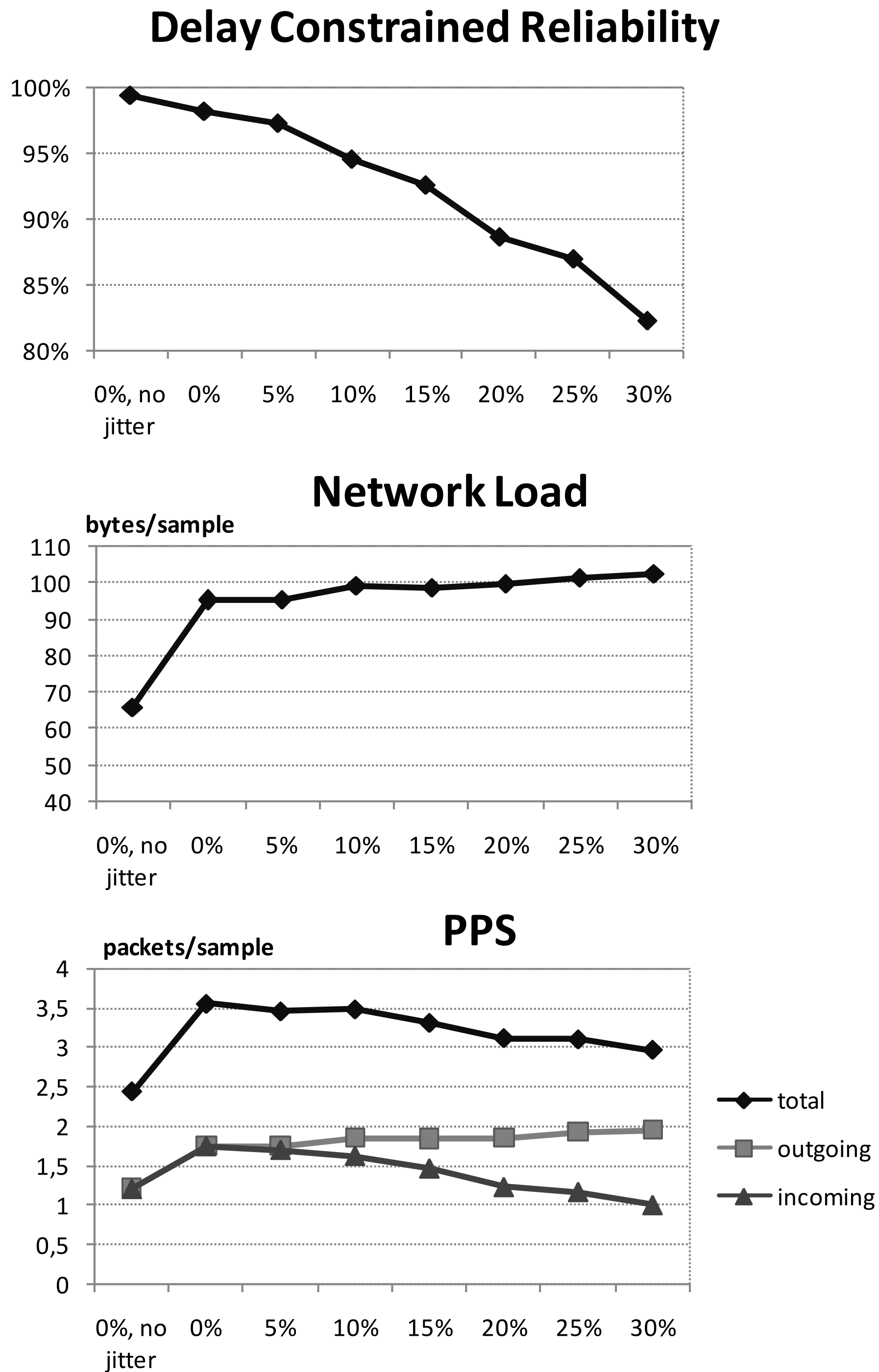

Figure 5.4: *Retransmission strategy at various packet error rates. The first coloumn is taken with no packet loss and no jitter according to the rule* `netem delay 75ms`*. All other columns are taken with the rule* `netem delay 110ms 80ms distribution pareto loss x%`

visual benefit of DCR increasing from 80% to 95% is evident. Playout at 80% DCR looks much more discontinuous, even though the general behavior of the subject is very clear anyhow. A tradeoff must be found also regarding power consumption. Higher sampling frequency and higher DCR imply a shorter device lifetime.

It is difficult to visually judge intra-stream synchronization. However, inter-stream synchronization is satisfactory. This is clear if one tries to drop the smartphone with the light and proximity sensors pointing to the target surface, so that they are fully covered after the impact. Vertical acceleration spikes at the same time as the proximity sensor becomes true and the light sensor reads zero lux. The simultaneous nature of the impact is very well represented, at least to a human eye.

Chapter 6

Conclusions

The outcome of this work consists of three main aspects (i) a demonstration by example that CoAP can fulfill requirements of real-time embedded Web services (ii) an open-source prototype as platform for future developments (iii) a demonstration that error correction techniques, in part offered by CoAP, are able to increment a set of application-independent Quality of Experience metrics.

The first point is based on the recognition of RTP as the state-of-the-art protocol for real-time multimedia applications. On this assumption, it is defined a precise mapping of the minimal set of RTP features to CoAP. The mapping is then successfully used in the prototype.

Considerable effort was invested in the server development. With the goal of optimizing the performance-footprint tradeoff, native Android development was chosen. The `libcoap` library needed additions to support CoAP Observe and to protect against packet duplication. The resulting implementation depends only on a standard POSIX-compliant C library so it is easy to port to less powerful nodes. It features a layered architecture to be easily extensible. The Streaming Manager can indeed be seen as an abstraction layer between stream physical sources and the communication protocol.

As mentioned, the concept of Quality of Experience depends on the application. In order to keep the analysis general it was devised a set of application-agnostic indicators: Playout Reliability to catch presentation quality and network

load to catch quality cost. The end-to-end delay upper bound for interactive application is included as a design constraint. In this regard, the synchronization algorithm is particularly important, because it sets meaningful deadlines accounting for inter-stream dependencies. In other words, it is the measurement instrument by which it is possible to assess QoE.

The practical tests have been performed on real devices connected by a simulated network able to reproduce worldwide area network characteristics. It consists of an Android phone connected via Wi-fi to a Linux host. QoS parameters such as network delay, jitter and packet loss are simulated by `netem` running on the Linux host. Despite slight modeling-induced errors, the results match expectations in all cases, thus proving the correctness of the implementation. They put in clear evidence the cost/performance tradeoff. Playout Reliability monotonically grows if error correction is turned on and if the network QoS improves. On the other hand, network load monotonically increases if error correction is turned off and the network turns unreliable. Thus, in an actual service deployment, given a set of fixed network QoS parameters, the designer can reach a certain QoE target by leveraging error correction mechanisms.

Future betterments may start by dropping the assumptions made here. For example, an algorithm to estimate the one-way average network delay would be useful to modify the buffering time upon changing network conditions. Also, this work achieves synchronization on the timestamp boundary. Adding sampling and playout hardware synchronization, i.e. outside the timestamp zone, would be beneficial.

In the context of constrained environments, a smartphone connected via LTE is a relatively powerful system. Further studies could concentrate on the challenges of porting this service to less powerful technologies.

At the application level, an important aspect is scalability. On the Web there could be potentially many clients interested in a device's stream. Also, these

clients could not be humans but machines. The best architecture, whether to use intermediary caches and brokers, and their influence on the timeliness of the communication could be object of further study.

Bibliography

[1] C.M. Aras, J.F. Kurose, D.S. Reeves, and H. Schulzrinne. Real-time communication in packet-switched networks. *Proceedings of the IEEE*, 82(1):122–139, 1994.

[2] O. Bergmann. `libcoap`. C library `http://libcoap.sourceforge.net/`.

[3] G. Blakowski and R. Steinmetz. A media synchronization survey: reference model, specification, and case studies. *Selected Areas in Communications, IEEE Journal on*, 14(1):5–35, 1996.

[4] C. Bormann and Z. Shelby. Blockwise transfers in CoAP. Internet Draft draft-ietf-core-block-14, 2013.

[5] M. Cesana. Internet of Things. Lecture notes, Politecnico di Milano, 2012.

[6] S. Firestone, T. Ramalingam, and S. Fry. *Voice and video conferencing fundamentals*. Cisco Press, 2007.

[7] Kouhei Fujimoto, Shingo Ata, and Masayuki Murata. Statistical analysis of packet delays in the internet and its application to playout control for streaming applications. In *IEICE Trans. on Communications, vol. E00-B, no. 6*, 2001.

[8] D. Guinard, V. Trifa, and E. Wilde. A resource oriented architecture for the Web of Things. In *Internet of Things (IOT), 2010*, pages 1–8, 2010.

[9] V. C. Gungor, Ö. B. Akan, and I. F. Akyildiz. A real-time and reliable transport (RT)2 protocol for wireless sensor and actor networks. *IEEE/ACM Trans. Netw.*, 16(2):359–370, April 2008.

[10] K. Hartke. Observing resources in CoAP. Internet Draft draft-ietf-core-observe-07, 2012.

[11] U. Hunkeler, Hong Linh Truong, and A. Stanford-Clark. MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks. In *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, pages 791–798, 2008.

[12] International Business Machines Corporation (IBM) and Eurotech. MQ telemetry transport (MQTT) V3.1 protocol specification. Technical report, 2010.

[13] Texas Instruments Inc. Android sensor porting guide wiki. `http://processors.wiki.ti.com/index.php/Android_Sensor_PortingGuide`.

[14] ETH Zurich Institute for Pervasive Computing. Californium. Java Library `http://people.inf.ethz.ch/mkovatsc/californium.php`.

[15] M. Kovatsch, S. Mayer, and B. Ostermaier. Moving application logic from the firmware to the cloud: Towards the thin-server architecture for the Internet of Things. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 751–756, 2012.

[16] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg. Implementation of CoAP and its application in transport logistics. In *Workshop on Extending the Internet to Low power and Lossy Networks (IP+SN 2011)*, 2011.

[17] V. Lampkin, W. T. Leong, L. Olivera, S. Rawat, N. Subrahmanyam, and R. Xiang. *Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry*. IBM Redbooks, 2012.

[18] N. Laoutaris and I. Stavrakakis. Intrastream synchronization for continuous media streams: a survey of playout schedulers. *Network, IEEE*, 16(3):30–40, 2002.

[19] Navid Nikaein and Srdjan Krea. Latency for real-time machine-to-machine communication in LTE-based system architecture. In *Wireless Conference 2011 - Sustainable Wireless Technologies (European Wireless), 11th European*, pages 1–6, 2011.

[20] E. Olson. A passive solution to the sensor synchronization problem. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1059–1064, 2010.

[21] C. Perkins, O. Hodson, and V. Hardman. A survey of packet-loss recovery techniques for streaming audio. *IEEE Network*, 12:40–48, 1998.

[22] Colin Perkins. *Rtp: Audio and Video for the Internet*. Addison-Wesley Professional, first edition, 2003.

[23] Steve Robbins. *Unix Systems Programming: Communication, Concurrency and Threads*. Prentice Hall Professional Technical Reference, 2 edition, 2003.

[24] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[25] H. Schulzrinne. Issues in designing a transport protocol for audio and video conferences and other multiparticipant real-time applications. Internet Draft draft-ietf-avt-issues-02, May 1994.

[26] H. Schulzrinne and S. Casner. RTP profile for audio and video conferences with minimal control. Internet RFC 3551, July 2003.

[27] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. Internet RFC 3550, July 2003.

[28] Z. Shelby. Embedded web services. *Wireless Communications, IEEE*, 17(6):52–57, 2010.

[29] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). Internet Draft draft-ietf-core-coap-12, 2012.

[30] C. J. Sreenan. *Synchronisation Services for Digital Continuous Media*. PhD thesis, University of Cambridge, 1992.

[31] R. Steinmetz. Human perception of jitter and media synchronization. *Selected Areas in Communications, IEEE Journal on*, 14(1):61–72, 1996.

[32] International Telecommunication Union. G.114 One way transmission time. ITU-T recommendation, 1993.

[33] J. Van der Meer, D. Mackie, V. Swaminathan, D. Singer, and P. Gentric. RTP payload format for transport of MPEG-4 elementary streams. Internet RFC 3640, November 2003.

[34] H. Wehbe, A. Bouabdallah, B. Stevant, and U. Mir. Analysis of synchronization issues for live video-context transmission service. In *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, pages 203–209, 2013.