Abstract

Popular music acts frequently rely on digital technology to help deliver their live shows. These artists use a combination of live instruments and prerecorded media, for example, recordings of instruments they cannot play on stage because they don't physically fit or because the artist want a specific sound difficult to reproduce.

The number of media files can be high; sometimes requiring more work than a single device can handle thus making it necessary to use multiple playback devices. To create a coherent experience, playback needs to be synchronized across devices to ensure they are all playing the right part of the file at the right time. Multiple playback devices can also prevent breakdowns by replicating the playback across several devices, which makes it possible to keep playing even if a device should fail.

Bespoke playback devices exist which can achieve the necessary levels of synchronization, but these tend to be very expensive and cumbersome to update when there are changes to the media. Commodity consumer computing devices, on the other hand, are inexpensive and easily updated because of greater software sophistication and connectivity. However, there is no distributed synchronization system that work on the types of computing devices and playback software commonly used by performers.

Through a series of interviews with stage performance experts, we have identified requirements for a distributed synchronization solution. Based on these requirements, and drawing from distributed systems literature, we present a solution running across Apples OS X and iOS devices. In principle, the system could further be extended to run on other platforms because the synchronization protocol is based on User Datagram Protocol (UDP), available on most platforms.

The solution has been verified through a series of tests performed in low and high network traffic scenarios. Our evaluation shows that our proposed solution is tolerant to device failures and can synchronize OS X devices within four milliseconds of each other. iOS devices have a slightly lower precision, being able to keep the devices within 11 milliseconds of each other. These numbers reflect real world performance because we measure at the audio output, opposed to only measuring the theoretical properties of our synchronization algorithm.

A Stanford study shows that musical performers need to be within 30 milliseconds of each other to perform well together. Measured against their findings our system is well below that threshold and therefore solves our problem. Our expert panel was also very pleased with our system finding its abilities and performance characteristics valuable.

Contents

List of Figures

Chapter 1

Introduction

Stage performers, such as popular music acts, rely on technology to execute their shows. Many artists use a combination of live instruments and prerecorded media, such as backing tracks. Humans play the instruments, whereas computing devices play the media files.

A performance comprises several parts meant to be executed in unison. To create a simultaneous execution of all the parts every performer needs to perform its own part at the right time. Giving the humans and computers on stage a shared notion of time gives them this capability.

Commonly, we use a linear progression of time called wall-clock time. Wall-clock time represents the elapsed, measurable, time passed from the start to the completion of a task. This simple approach provides a strong foundation for the problem we are targeting. The actors on stage only need to know time in relative amounts. How much time has passed since the beginning of the song, the beat, the last event? Wall-clock timing provides an answer for all these types of questions.

If the actors on stage did not have a shared notion of time, they would have no way of combining their efforts into a uniform piece. Essentially the performance would be an uncoordinated mess where each actor would perform their own piece of the performance at a random point in time.

1.1 Synchronization

The basis of a coordinated performance is that all participating actors know when they should perform their parts, they must be synchronized. Musical artists are used to executing their actions in sync with other performers. If we take a typical band we have a drummer and some melodic performers. The drummer beats the drums creating audible reference points for the rest of the band to synchronize their actions to. Musicians synchronize their actions by listening to reference points, be it drums beating or any other audible reference. In this section, we discuss the notion of time synchronization and explore the different forms it can take.

1.1.1 Human Synchronization

Human actors use their senses to synchronize actions to other actors. A drummer can synchronize a band by counting them in. A conductor synchronizes an entire orchestra by swinging a baton. Both cases involve a metronome-like synchronization based on musical beats. These approaches are less concerned with synchronization of absolute clock time (e.g. that all performers know that it is 20:00:00), but rather with the synchronization of the musical beat.

1.1.2 Computer Synchronization

Computers, unlike humans, are good at using absolute timing. An absolute timestamp makes it easy for a computer to jump to that position in a media file. The commodity hardware we are targeting is laptops and hand-held devices. We will use the built in communication capabilities, namely the network stack, of these devices to synchronize time. On laptops this includes Wi-Fi and Ethernet connections, on mobile devices this include Wi-Fi. However, there is a major problem when using the network for synchronization; the time it takes for a message to go from a sender to a receiver is non-deterministic. To create a good synchronization we need to create an algorithm that tackles this non-deterministic delay.

1.2 Systems, Devices, Computers and Nodes

From this point onwards, a system will mean the entire technological solution for synchronizing a stage performance. This can include everything from one to multiple devices. A device is the specific hardware unit, normally a laptop computer. When talking about the specific device or computer, we will use the terms device, computer, phone, tablet or laptop. When talking about a device in relation to the system, the term 'node' is used. A computer is a device if it is not part of any system, and when it is part of a system, it is a node in that system.

1.3 Why Multiple Devices?

The devices available to play media files have limited resources. The bandwidth of the storage medium sets the upper limit of concurrent media files a device can play. Some performers process the media before it is audible, and processing use CPU cycles making the processor a limiting factor. Device memory is the limiting factor of the number of files that can be concurrently loaded into memory. These limitations combined create an upper limit of how many files a single device can handle.

If you need to playback more files than one device can handle, you need to spread the workload on multiple devices. However, a multi-device setup creates a communication problem; when should the playback start and stop? In a single device setup, you just press play and stop at the correct time; however, multiple devices have no shared notion of time. In our literature review, starting on page 5, we will look at examples from the

research community on creating a shared notion of time by synchronizing the clocks in each device.

1.4 Computer Roles and Reliability

Reliability concerns must be addressed before bringing computers on stage. What should happen if a computer malfunctions during a show? Surely, the performer wants this to go unnoticed by the audience, creating a need for a backup to be in place. Introducing a backup device necessitates keeping this device synchronized in order for fall-overs to happen with as minimal interruption as possible.

Since computers are versatile, they can assume different roles in a stage performance. A computer can have either real-time or time-dependent behavior. The two roles have very different solutions to the problems in an on-stage environment.

Real-time systems process messages as they arrive from an input (such as a MIDI port) and produce an output, ordinarily an audio output. This could be a software-based synthesizer that responds to input from an external physical controller. Redundancy is easy to achieve in real-time systems, sending the same messages to several nodes is enough to solve the problem. This works because each node is an isolated unit that need not know anything about other nodes; it takes the incoming message and transforms it to an output as fast as possible.

Time-dependent systems fire events at a specific point in time. This could be media playback, where a performer is controlling the playback using external control. External control can be achieved by using a time code generator, which sends timing information in a steady flow to the receivers. This approach is called a master / slave setup, where the time code generator is the master that dictates where the slaves should play. The nodes of a system will not drift apart when running of an external clock, such as a time code generator. Figure 1.1 shows a master / slave setup. A fundamental flaw with this approach is that it creates a single point of failure, the master. If the master fails and stops generating time code then the typical behavior of a slave is to stop.

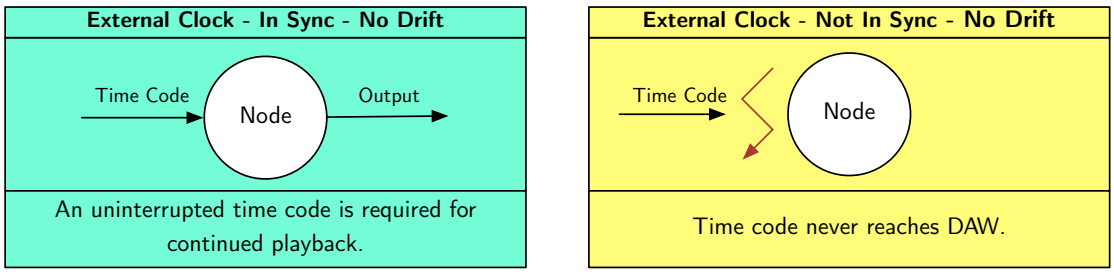


Figure 1.1: A master / slave setup where the master is providing time code. The playback will start as soon as time code is received. However, the Playback will stop if the time code is interrupted.

For stage performances, it is not acceptable to have a single point of failure, other solutions are needed. A commonly used approach is to only synchronize the start of the playback across the nodes of a system and have them run on their own internal clock.

This is a more reliable solution because nodes are not dependent on other nodes once they are playing, and if a node breaks down the rest will be unaffected.

A typical setup used by performers today is one where two or more computers are triggered by a single external controller. The external controller can be a MIDI keyboard or a touch screen controller, the output of the controller is routed to the input of each computer. The controller can send playback commands to the connected computers. Figure 1.2 shows a system where each node is using the internal clock and receiving play commands from an external controller.

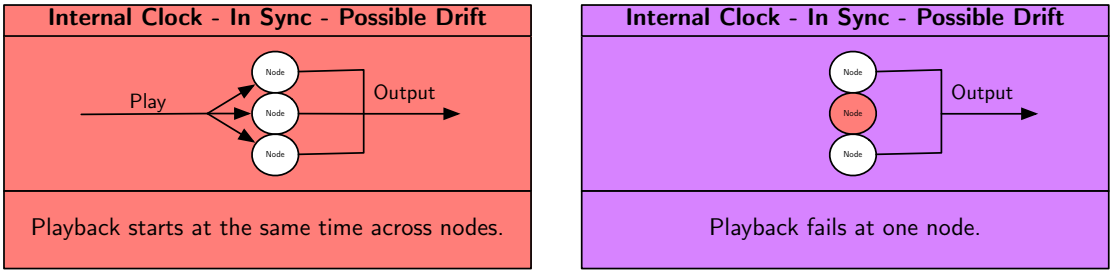


Figure 1.2: Playback commands are needed when an internal clock is used. The playback will start when the play command is received. On the right, you see a system where one node has failed, but the rest is unaffected.

The biggest flaw in the last approach is that the nodes will drift away from each other as they play. The internal clock in a node will drift over time, which causes a clock offset between each node in a system. The consequence of this is that the output from each computer will not stay synchronized, the longer a system is left playing the less precise it will be.

1.5 Clock Drift Analogy

Imagine going into a room closing the door and counting to 1000. You should come out of the room when you reach 1000. If you were to do this in a corridor along with other people, the chance of all of you coming out of your doors at the exact same time is tiny. This same thing happens for computers. Each computer has an internal clock, typically a quartz clock. This clock runs at a certain rate, based on the oscillations from the quartz crystal. This rate is not the same across devices, even when the devices are the same brand and model. Each quartz crystal is unique and has its own oscillation rate. Just as humans, computers count at different rates.

1.6 Summary

In this chapter, we have presented the need for mixing human and computer performers on stage and with it the problems that incur. We talked about the limiting factors of computing devices, which can create a need to use multiple computers. Multiple computers need to be synchronized, and we have looked at two ways of synchronizing multiple devices, either with an external clock or with an internal clock.

Chapter 2

Literature Review

Our aim with this thesis is to solve a problem for stage performers, the problem of how to achieve safety and coherency for performers who rely on multiple computing devices in their work. When multiple computers work together they form a distributed system, and these systems has the advantage that additional capabilities or more processing power can be added by connecting more nodes to the system.

Problems arise when a distributed system need to coordinate its actions across several nodes because each node in a distributed system is a self-contained unit and cannot directly use the resources of other nodes. The common way for nodes to work together is by message exchanges, generally communicated through the network stack.

Stage performers need to have their system synchronize actions such as playback of media files across all nodes to achieve the coherence needed for a good performance. To achieve synchronized behavior across nodes we need them to have a shared notion of time, they must all have synchronized clocks. To get a better understanding of these types of system the overarching category for this literature review will be the research field on distributed computing and most of the papers is in a subcategory focusing on clock synchronization.

Clock synchronization has been the subject of extensive research ever since the advent of distributed systems. The goals of clock synchronization is to compensate for offsets and drift between clocks. Offset is how far the current clocks are from each other. One clock can show 12:00 another 12:05 this would be an offset of five minutes. Drift is when clocks don't run at exactly the same speed, they count time at different rates. If we have two clocks show 12:00, and we wait 24 hours we would expect both clock to show 12:00 again. However, because of the drift there could be a difference so that one clock might show 11:59 whereas the other would show 12:01. The goal of clock synchronization is to correct both types of differences and have the clocks read the same value at all times.

Not all synchronization methods require message exchanges between the nodes of a system. It is possible to have each node synchronize their own clocks to an external clock, like an atomic clock. Atomic clocks are extremely precise; modern atomic clocks lose around one second every 13.8 billion years [14]. Nevertheless, atomic clocks are as costly as they are precise, making them too expensive for most systems.

Another external clock, which can be used is the Global Positioning System (GPS) that use three atomic clocks in each GPS satellite. Because the atomic clocks are very

accurate and almost doesn't drift it is possible to calculate positions around the globe using triangulation calculations against three satellite signals.

Since the GPS system is based on a very precise timing system, it is possible to use the GPS system as an alternative time reference. GPS can be used as a timing reference by equipping a device with a GPS receiver. Such a receiver is much cheaper than an atomic clock, making a GPS based solution more cost effective. However, a GPS receiver requires a direct line of sight to the GPS satellites making it unusable indoors and since many performances happen on indoor stages, a GPS based solution is not applicable for our problem domain.

It is also possible to synchronize clocks without all nodes having direct contact with an external time reference; nodes can communicate time using message exchanges. However, if the system needs to be synchronized to a universal time format such as Coordinated International Time (UTC), at least one node must have contact to a timing reference providing this time format.

Some systems need not be synchronized to a universal time format, instead using an internal time format where nodes are synchronized in relation to each other. Leslie Lamport, winner of the 2013 Turing Award[1], has created the foundations that make such a system possible. In his seminal paper on virtual clocks, Lamport described how to order events in a distributed system [5]. Lamport formalized the fact that send events must happen before receive events, such that if node i send a message to node j, we know that the send event must have happened before the receive event. Building on top of these relative relationships it is possible to create protocols for absolute timing relationships.

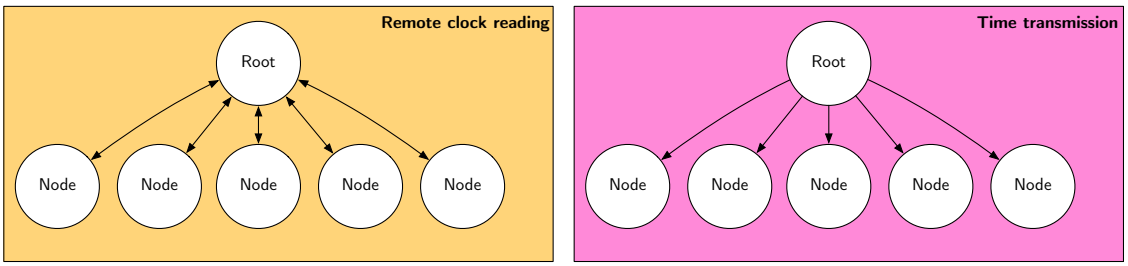


Figure 2.1: Two common techniques for synchronizing clocks; remote clock reading is where a node asks a root for its time, time transmission is when a root periodically sends its time to receiving nodes.

Many protocols have been designed which can synchronize absolute time values [3, 6–9, 12, 13, 15, 16] A common feature of these protocols is that they synchronize clock information using the network infrastructure itself, no additional external hardware such as an external clock is required.

There are two common ways of doing these message exchanges, either by remote clock reading or by time transmission, both are illustrated in Figure 2.1. Remote clock reading is when a node asks the root what the value of the roots clock is and then adjusts its own clock accordingly. With time transmission, the root continuously sends its own clock reading to all connected nodes.

[1] `http://amturing.acm.org/award_winners/lamport_1205376.cfm`

All communication takes time and so does message exchanges over the network. This is the biggest obstacle for perfect synchronization of computer clocks because you don't know how long it takes to exchange a message using the network stack. So when a node asks for the roots clock reading it does not know when the root did the reading compared to when it received the message containing the reading. These non-deterministic packet delays in message delivery and processing incurred by the local and external networking infrastructure is something that all synchronization protocols based on message exchanges has to handle. The parts of the packet delay that is incorporated into a given protocol is called "the critical path".

Bharath et al. decomposes the non-deterministic message delivery delay into four parts; send-, access-, propagate- and receive-time [10]. First, the sender spends time building the message. Second, the sender needs to put the message on the network interface card (NIC) and await that the NIC has a clear channel to send the message on. Third, the message takes time to propagate throughout the network. Fourth, the receiver spends time receiving and processing the message. This division of the message delivery delay is shown in Figure 2.2 along with typical real world bounds of the time it takes for each part of the delivery to complete.

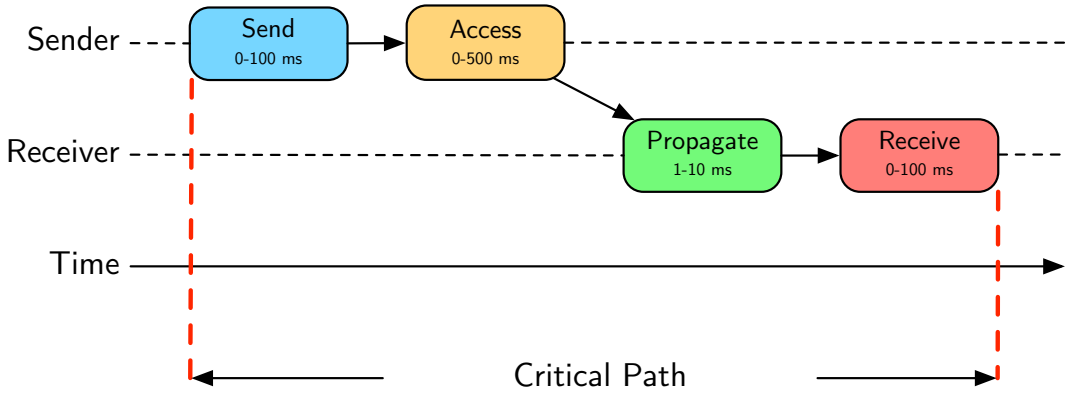


Figure 2.2: Decomposition of packet delay in a network with typical delay sizes shown with a critical path covering all four stages of the packet delay.

Flaviu Cristian presents a way to minimize the problem of the non-deterministic delays by calculating the minimum delay in the network [2]. Cristian argues that you can estimate the delay to any node within a known bound; you measure the delay several times until the readings form a normal distribution. Experiments described in his article has shown that most networks have a delay distribution similar to the one in Figure 2.3. This distribution shows that most messages have a delay, which is close to the minimum delay. The infrequent messages with a much longer delay can by discarded making it possible to create a precise synchronization based on remote clock reading.

One of the early, and still widely popular time synchronization protocols is Network Time Protocol (NTP) [8]. NTP works by having its clients synchronize their clocks to NTP timeservers using remote clock reading. The NTP algorithms minimize the non-deterministic network delays by doing statistical analysis of the round-trip time incurred by the remote clock read. NTP is synchronized to the external time format UTC and the

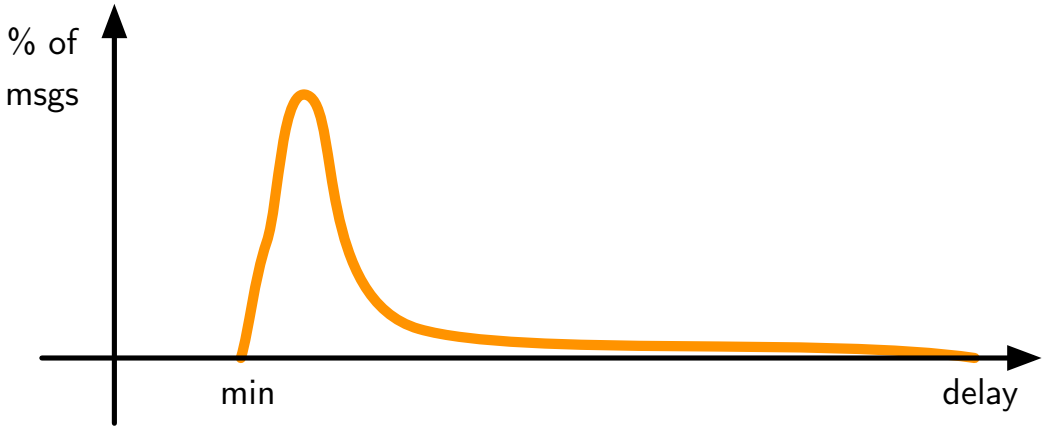


Figure 2.3: Normal distribution of network delays.

timeservers use external time references, often a GPS receiver. NTP differs from many of the other protocols because it was designed to work in large-scale networks. NTP is arguably the protocol with the most widespread adaption as it is used on most platforms available today. Because the NTP software is available on so many platforms, it is easy to setup and incorporate into any system. NTP is open source software, making it a transparent and obvious choice for many applications. The problem with NTP is that it is not very precise; a typical NTP setup will have an accuracy of around 10 milliseconds. Newer versions of NTP can achieve better accuracy by using a modified network kernel.

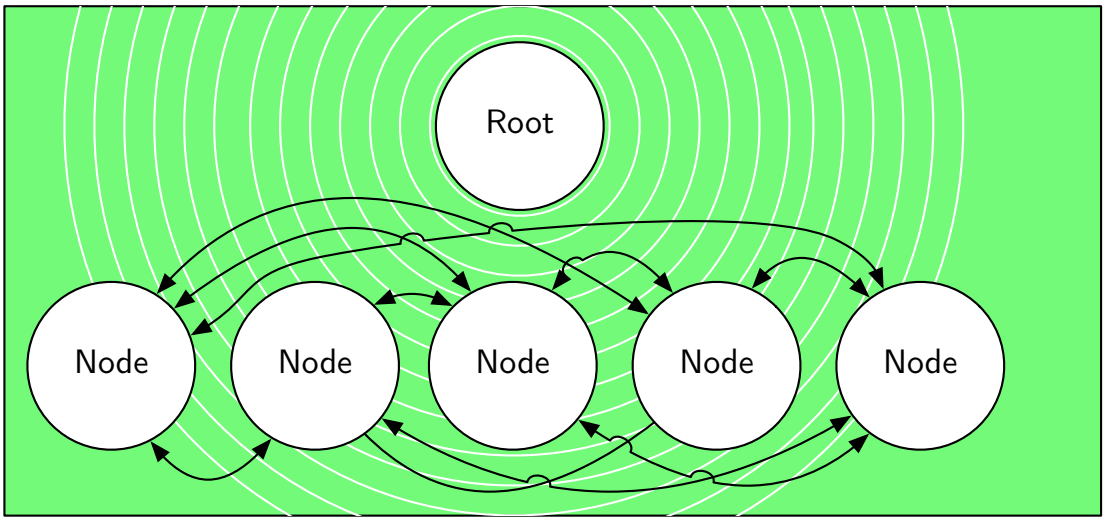


Figure 2.4: Reference broadcast synchronization works by having a root node send out pulse messages, which are observed by receiving nodes. Each node records the time it received a specific pulse and exchanges this information with all other nodes. Because there is almost no divergence between reception times this is used to synchronize the clocks of the nodes.

Elson et al. presents an alternative take on time transmission, the reference broadcast synchronization (RBS) [3]. Their work builds on the fact that the propagation time of broadcast messages is close to zero [15]. The propagation time for nodes within 300 meters from a router is under one μs (microsecond - 1/1000000 of a second). RBS uses a dedicated root that broadcasts pulse messages which is then received by all nodes at the same time. After each client has received a message, they exchange their local time of reception with the other nodes. Because all nodes receive a pulse at the same time this can be used as a reference point for calculating the offset between each node.

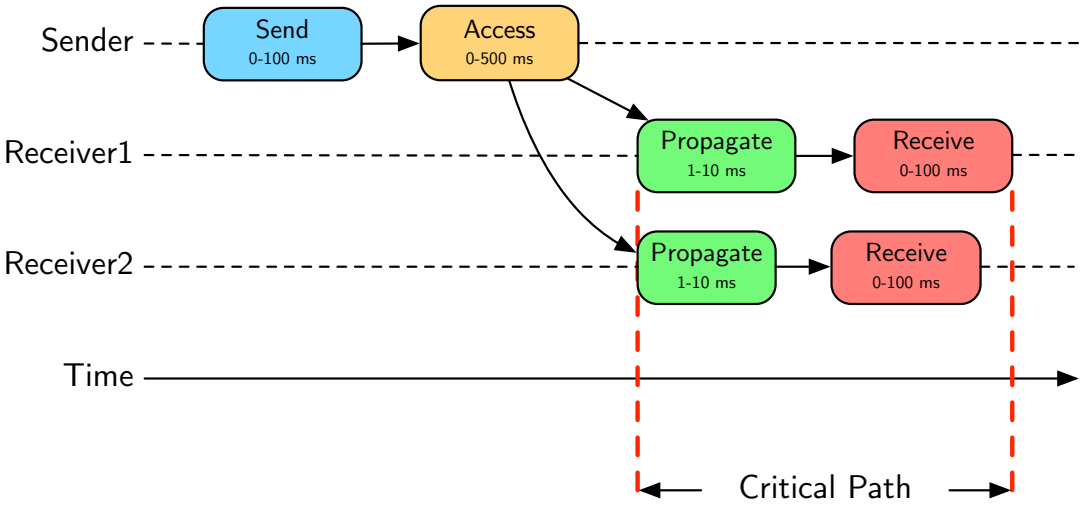


Figure 2.5: The reference broadcast synchronization has a smaller critical path because it eliminates the send- and access-times from its algorithm.

The constructing of their algorithm lets them remove the send- and access-time from the calculations. By removing the largest unknown delays they minimize the critical path as illustrated in Figure 2.5. RBS is on average 8 times more precise than NTP, but it does not scale to the same size networks that NTP can handle.

The rise of sensor networks has driven much of the research on clock synchronization in recent years. A sensor network is comprised of resource-restrained devices, where battery and processing power is scarce. Therefore, the focus of many researchers has been how to achieve clock synchronization with a minimal impact on the sensors resources. RBS uses a high amount of message exchanges to achieve synchronization as each node communicate observations with all other nodes. On a network of n nodes, RBS uses $O(n^2)$ message exchanges; this can take up a large amount of the resources in a node.

Ping argues that the RBS protocol can be improved by lowering the number of exchanges needed for synchronization and introduces a new protocol called Delay Measurement Time Synchronization (DMTS) [9]. DMTS exploits two benefits from running on the real-time operating system TinyOS[2]; it allows for fine-grained control over the network stack and it allows for very precise timestamps. This makes it possible to await a clear transmit channel before creating and transmitting a timestamp, thereby also minimizing the critical path to propagation- and receive-time similar to RBS. DMTS requires

[2] `http://www.tinyos.net/`

low-level access to the kernel, for precise timestamp, and to the NIC for being notified when a clear channel is available. This makes DMTS much less precise on non-real-time operating systems such as OS X and iOS.

Sometimes sensor networks are large, requiring several routers in a multi-hop configuration. RBS is not an ideal protocol for such setups because it loses precision as the network adds more hops. The Timing-sync Protocol for Sensor Networks (TPSN) [13], Flooding Time Synchronization Protocol (FTSP) [7] and Gradient Time Synchronization Protocol (GTSP) [12] try to improve the precision on larger networks. They achieve this by building and maintaining a structure of the nodes and having each node synchronize to neighboring nodes. Their actual synchronization algorithms is a combination of remote clock reading and pulse broadcasts. They all need low-level access like DMTS to timestamp packets at the Media Access Control (MAC) layer, making them less precise on a non-real-time operating system which does not have API access to this layer.

A very different approach to synchronization is introduced with the Reachback Firefly Algorithm (RFA) [16]. RFA is inspired by how fireflies synchronize their flashing lights. RFA is a fully distributed synchronization algorithm where all nodes share the same behavior. Nodes generate a pulse message and observe pulses from other nodes to adjust its own firing rate. If a node receives a message before it was about to send one itself it will shorten its time before sending its next pulse. After a while, all nodes will pulse at the same rate and thereby having synchronized the clocks. The advantage with RFA is that it is an algorithm, which is easy to understand and implement. A disadvantage is that it can take a long time, up to several minutes, before all nodes of a system is coordinated. In addition, for RFA to be implemented with network exchanges it, like the previous protocols, also requires low-level access to the MAC layer.

Ridoux et al. presents an approach to improve timestamping accuracy on a commercial operating system. They patch the Linux and Free BSD kernel to get access to the most precise timers of these kernels [11]. With their kernel patches, it would be possible to implement all the above protocols on some non-real-time operating systems.

The aim of our work is to make it usable for stage performers, so we need to know how sensitive these performers are to offsets in time. The speed of sound, 340.29 m/s, has a large impact on the ability to perform coherently. Research by Gurevich et al. concludes that the distance between performers must be below 10 meters, similar to 30 milliseconds, for performers to keep the same tempo [4].

The study had pairs of test subjects clap a simple rhythm, the aim being they should clap the same rhythm and keep a stable tempo throughout the test. The subjects were placed in separate rooms such that the researchers could control the delay of the signal they could hear of the other subject clapping. The tempo and precision of the claps was stable until the signal was delayed by more than 30 milliseconds. Above 30 milliseconds, the tempo dropped and the precision, the interval between each clap, deteriorated as the delay increased.

Chapter 3

Analysis and Requirements

In this chapter, we will interview three experts. All working on or around the stage of professional music performances. We will analyze their interviews and use this for building the requirements for our own work.

3.1 Interviews

We have interviewed three experts from the professional performing music scene. One is a manager for large international acts, another is a performing musician in several international acts and the last expert is a founding member of an experimental laptop orchestra.

3.1.1 Interview Method

The interviews took the form of a natural conversation, around one hour long, situated in a relaxed environment; we interviewed expert one at his office, expert two at a café, and expert three in his home via Skype video chat. Each interviewee answered a series of fixed questions. These questions had the purpose of understanding several unknowns about synchronization. Is synchronization a problem? Why is it a problem? What happens if you do not synchronize?

To avoid the experts influencing each other's answers the interviews were conducted separately. The answers were recorded using an audio recording and written notes. The following sections will summarize each interview. Lastly, we will combine the similarities from all experts into a summary of synchronization problems.

3.1.2 Expert One – The Tour Manager

This expert has the responsibility for large installations, which include several devices that need correct time. This includes video-, lighting- and audio-playback devices. These devices must play at the same time in order for the performance to have its intended effect. Synchronization can be a problem in at least two ways; a device can get confused and play at a wrong time causing a chaotic performance, or it cannot play at all making only a subset of the performance available.

This expert has used the Society of Motion Picture and Television Engineers (SMPTE) time code in several setups. In these setups analog SMPTE time code, running on its own cable, synchronize the time sensitive devices. Getting the analog time code from A to B has some limitations because it is transmitted on analog cables. First, the signal source must be clean, meaning that the audio must have a good signal to noise ratio. There must be more signal than noise for the time code signal to be readable. Second, the signal needs extra analog cables running between the devices that need to sync. Third, the devices that need to read the time code must have audio input capabilities for it to receive the time code signal. Finally, analog cables are not perfect; they are susceptibility to external noise. If you put them nearby power cables or other sources that emit noise that noise can go into the cable. This noise will then pollute the time code signal and possibly corrupt the timing information. If the signal gets too noisy, the receiver can have problems understanding the time code, making the synchronization fail.

Unbalanced analog cables are especially subject to this problem because they have little protection against external noise. Balanced audio cables, however, cancels out the noise that the cable picks up through phase manipulation. Nevertheless, in the real world, it is not always possible to run a balanced signal end to end. Sometimes the cable needs to run through a series of converters to fit the connectors of the participating devices.

SMPTE time code has a digital adoption in MIDI called MIDI Time Code (MTC). This enables that one cable can have several uses because it can share signals. One MIDI cable can carry time code along with musical- and control-messages. This expert has two problems with MIDI on stage; few of the devices needed on stage understands MTC and MIDI cables work best at lengths under nine meters, making it hard to use MIDI cables on large stages.

Another problematic aspect with synchronization is setup. The typical equipment used in stage performances are mainly musical instruments. Technicians handle these instruments, drums by a drum tech, guitars by a guitar tech etc. The computing devices are typically handled by a keyboard tech, but according to this expert keyboard techs are a limited resource compared to the other tech types. This is a problem since many acts are using computing devices. Simpler systems could make it easier for more technicians to setup and maintain the systems.

Finally, this expert finds that the expense of a reliable synchronization setup can be a problem. Today a good solution can be costly, and it is not all acts who has the budget for today's systems. This creates a need for a cheaper solution.

3.1.3 Expert Two – The Musician

This expert is a keyboard and percussion player for several international music acts; including some where musicians play together with prerecorded media files. One or several devices handles the playback of the prerecorded files. Most of these files are intended for the audience but some are only intended for the musicians to hear, such as a click track. This click track makes sure that the musicians on stage can play along in the same tempo.

According to this expert synchronization is a problem if you need to play music at the same tempo. Some venues provide poor acoustic properties for the musicians, making it hard for them to pick up the finer details of the music they are playing. Musical expression

ranges from soft to loud, and if you are on the soft end, it is not always possible to feel or hear the music in a challenging acoustical environment. Having the musicians follow a click can solve this problem, since they now share a common time reference.

This expert feels that playing to a click allows for greater creative freedom in the performance, you can have longer pauses between notes and still hit the right beat when rejoining the performance. The click need not be an audio signal; it could also be visual cues. The important aspect is that the click must be stable and it must tick at the same time for all musicians on stage, making it possible to follow the same tempo.

If events on stage are unsynchronized, you cannot talk about it being a combined performance. It will turn into a chaotic uncoordinated performance. A poorly synchronized device on stage resembles having a musician that cannot play.

One of the worst things that can happen if synchronization fails is that the audience will notice. If you play a steady tempo and suddenly this changes because of synchronization failures it can lead to confusion amongst the audience and the performers. This is an undesired event, which can kill the magic feeling of an otherwise good performance.

The main reason for having multiple playback devices, for this expert, is to make sure that the show can keep playing in the event of a device failure. If one of the playback devices fails, you can switch the playback to another device immediately. To achieve this the devices need to playback in perfect sync with each other.

This expert has also experienced the same lack of keyboard techs as our first expert.

3.1.4 Expert Three – The Digital Pioneer

This expert feels that the more we can do with different devices the more essential it gets that these devices can synchronize their actions. However, whereas device capabilities have grown exponentially over the years, the technology to synchronize them are lacking behind. This creates an individualistic environment and not a collective one where multiple performers can be creative together.

Synchronization is important in music because playing music is about communication, communication between the performers and the audience. If there is no synchronization this communication gets lost and the performance gets noisy and chaotic. When synchronization goes wrong, we lose confidence in the computing devices. Synchronization problems make the performers uncertain about their actions; they lose their foundation of execution. It shifts the focus from performing to problem solving technical issues, which have little to do with a musical performance.

This expert has developed a method to synchronize several Ableton Live[1] devices through a homegrown Max[2] patch. This approach is sufficient for the needs of this expert. Before the Max solution, this expert used a simpler approach to synchronization, where each member of the band pressed play at the same time. Nudging the tempo in Live can correct any small offset that may come from using this approach.

There are a few limitations to this experts Max solution. First, it is a master / slave setup, meaning that if the master breaks down the clients will stop playing. Second, it needs to run on a cabled network connection to be stable. The expert has tried using it on a wireless connection, but this caused too many problems to be usable.

[1] `http://ableton.com/live`
[2] `https://cycling74.com/products/max/`

3.1.5 Summary of Interviews

Expert one has two main problems. The first is the physical limitations of analog and MIDI cables. The second is that the setup gets more complicated the more synchronized computing devices you use and the people who can handle these setups, the keyboard techs, are hard to find.

Expert two echoes the problem that the setup procedure can be cumbersome and that keyboard techs are a scarce resource. The main problem for this expert is that synchronization needs to be reliable and tight to create the desired experience for the performers and the audience.

Expert three echoes the problem that synchronization is vital for creating the wanted experience. Not only is it vital for the performance, but it is vital for creating a high level of trust between human and computer performers.

The experts voice similar concerns. They share the need for a reliable and tight synchronization between devices. The third expert has defined tight synchronization to be 480 ticks per beat. The other two experts have a more loose definition, one where performers and audience does not experience differences between devices.

The first two experts finds that setups involving synchronized devices are harder to setup and maintain than regular musical setups where only normal instruments are used. This is not a problem for our third expert because of the homegrown solution to synchronization.

Based on the interviews there is a need to create a tight, reliable and easily deployable synchronization solution.

3.2 Requirements

In this section, we will create a list of requirements that our solution should fulfill. From the interviews, we concluded that there is a need for an easy to use synchronization protocol. The three experts mainly use Apple laptops, mimicking our own experience that Apple is the dominant provider for stage performers. Creating a solution that works on Apple laptops is a high priority.

Based on our own experience we know that Apple hand-held devices are popular with stage performers. The main usage today is remote controlling laptops off stage. However as these devices get more powerful they become more interesting as playback devices too. Our work should have this in mind when creating the solution.

The synchronization itself, how tight the devices should be, should be better than the 30 milliseconds limit found in the research by Gurevich et al. [4]. Their study found that performers were able to perform together if the delay between them is under 30 milliseconds. This is exactly what we wish to achieve, we want to make it possible for human and computer actors to perform together.

The solution architecture should try to address shortcomings from the traditional master / slave setup. Traditional setups have the problem that when a master malfunctions it is highly likely that the clients will do the same. If a master is generating time code, and the cable is cut or the connection otherwise dropped then every client will usually stop. We want a solution where node dropouts will not affect the remaining nodes.

The experts talk about ease of use being a high priority for them. We conclude that this means that the setup does not require additional, or special, cabling. We will use the networking capabilities of the hardware they already use. Besides cabling, we understand it as a need for a reliable and reproducible setup procedure. The solution should be easy to start and have predictable behavior on every startup. If there are any problems during setup, presenting them in a way that is easy to understand is a requirement.

3.2.1 List of requirements

This is a list presentation of the requirements identified above.

1. Support OS X targeting Apple laptops computers.
2. Support iOS targeting Apple hand-held devices.
3. Node dropouts must not affect playback of other nodes.
4. Easy, reliable and reproducible setup procedure.
5. Synchronization tightness of 30 milliseconds or below.

3.3 Summary

In this chapter, we interviewed three experts and used their knowledge to identify a list of requirements for our design and implementation phase, which is coming up in the next chapter.

Chapter 4

Design and Implementation

In this chapter, we will look at the process of designing and implementing our solution. First, we will explain the exploration phase where we researched different technological approaches. Finally, we go into the specific implementation, how we built our solution with the chosen technology.

4.1 Technology Exploration Phase

Creating an application can be done in many ways using an increasing number of programming languages and frameworks. Initially we thought it was feasible to create a cross-platform application with one fully shared code base. To test if our thoughts could be realized we did some prototyping with two different technologies; Xamarin and Swift.

Xamarin

One way to create a cross-platform application is by using Xamarin[1] that promises that you can target multiple platforms including Windows, Windows Phone, Android, OS X and iOS from a single code base written in C#. Xamarin is built on top of the work by the Mono project[2], which is to bring the Common Language Infrastructure and Common Language Specifications, known as Microsofts .NET, to Android, iOS and OS X. Xamarin makes the Mono project more approachable by providing tooling support and a way to create app store ready applications. However, whereas the Mono project is open source, the additional functionality in Xamarin is closed source and requires a paid license per platform you want to target.

Xamarin gives you two options for code sharing across platforms, a shared code project or a Portable Class Library (PCL). If you choose a shared code project you end up shimmering your code with if / thens to have different code per platform where needed. PCL however targets only a subset of the .NET platform available on the platforms you target. When you have selected the targets you need, you are presented with a choice of a PCL profile. Depending on which profile you select you have a different subset of .NET available to your application. The least restrictive profile I could find which targets iOS, OS X, Android and Windows is profile 78. This profile, as all the others available, sadly

[1] `http://xamarin.com/`
[2] `http://mono-project.com/`

does not support sockets, which is our intended way of communicating between nodes in our system. Because we couldn't use sockets with a PCL we felt rather unsatisfied by the Xamarin offer. Although it would be possible to use sockets with Xamarin this would require a separate code base for each platform and this is what we wanted to avoid.

After our initial tests with Xamarin, we put it on hold until we knew exactly what type of application we needed to build. When we did our interviews, we gained more knowledge that led us to the fact that maybe Android and Windows is not the primary targets for our audience. What we know is that we need to build a distributed application that can synchronize its nodes and is tolerant to failures. We also know that it should, at least, support the Apple operating systems, as these are the major platforms identified through our interviews.

Xamarin can be used for targeting Apples platforms but it treats iOS and OS X as two different platforms, thus needing to use two APIs and two code bases. Xamarin is on the verge of releasing iOS and OS X unified APIs, which will make it easier to share code between iOS and OS X. We tried an early release of this, but unfortunately, it was too unstable when we tested it.

We needed another approach, and we turned our attention to the newly released programming language Swift.

Swift

Swift is Apples new programming language, created by Chris Lattner[3], introduced in the summer of 2014 and it allows for building applications for OS X and iOS from one code base. Although Swift supports fewer platforms than Xamarin, the narrower scope eliminates some of the shortcomings of Xamarin. With Swift, we could use sockets across our platforms from one code base, which is very helpful in keeping the code smaller and more maintainable. Swift also targets OS X and iOS from many of the same libraries, which has led to almost 100% code sharing between the two platforms.

Many of the protocols presented in the literature review use hardware level timestamping of network packets, where messages are timestamped as soon as they arrive at the NIC. This is possible because the protocols run their solutions on real-time operating systems such as TinyOS, and the TinyOS developers has put a lot of work into creating a fast and lightweight operating system that can perform in real-time scenarios. Apples non-real-time operating systems iOS and OS X does not provide an API for getting timestamps at this level so we can't use hardware level timestamping.

What we did instead was a slight modification of an open source socket library called CocoaAsyncSocket[4]. CocoaAsyncSocket has UDP and TCP sockets in different implementations. One of these implementations is built atop Grand Central Dispatch (GCD), which is a thread pool sharing API from Apple that allows quick parallel code. Our modification to the library has made it possible to timestamp incoming messages before they are sent through GCD to the receivers by a delegate function. This allowed us to build a tighter synchronization algorithm because we get a slightly more precise timestamp of incoming messages compared to the standard version of CocoaAsyncSocket.

[3] `http://nondot.org/sabre/`

[4] `https://github.com/robbiehanson/CocoaAsyncSocket`

Because Swift is a young language we feared that we couldn't use third party libraries, and that we had to write everything from scratch ourselves. This turned out not to be a problem because of Swifts interoperability with Objective-C and C. All the libraries that already exist in Objective-C and C can be used directly from within Swift, the socket library, CocoaAsyncSocket, we used is in fact built with C and Objective-C.

Apple is trying to get all developers to prolong the battery life of their devices by enforcing power saving on OS X and iOS applications. When you create an application for these platforms, they will by default try to put the application to sleep when Apples algorithms detect that the application is not in use. We experienced problematic behavior with these algorithms as they put our application to sleep when it was still in use. Luckily, you can override the default power saving settings by disabling *AppNap* on OS X and the *idleTimer* on iOS.

Although we only used Swift as a test, we haven't felt the need to explore further technologies. In fact, our code has just evolved from the early prototyping to what it is today. We have found that Swift is the ideal choice for our implementation because it is easy to use and it runs from a single code base across iOS and OS X. The only code we don't share fully between the platforms is the user interface code, which in our case is little code.

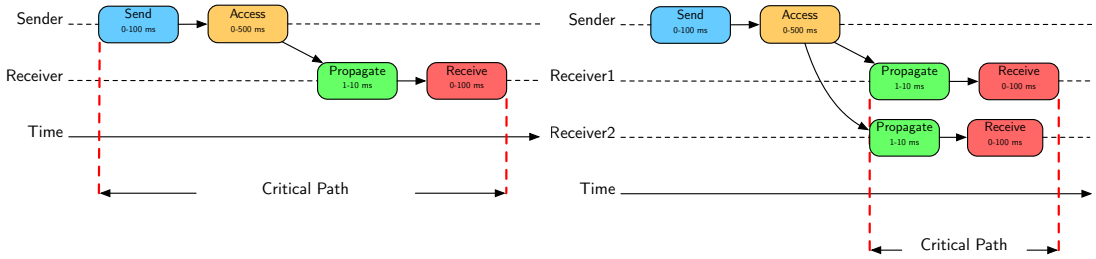
4.2 Implementation

After our technology exploration phase, we began our actual implementation. Our aim with the implementation was to fulfill all the requirements we identified in Section 3.2:

1. Support OS X targeting Apple laptops computers.
2. Support iOS targeting Apple hand-held devices.
3. Node dropouts must not affect playback of other nodes.
4. Easy, reliable and reproducible setup procedure.
5. Synchronization tightness of 30 milliseconds or below.

4.2.1 Synchronization Algorithm

To achieve our goal of getting multiple nodes to play simultaneously we need to synchronize the nodes, and we have looked at several interesting approaches in the literature review. Many of these clock synchronization protocols share a vital component; the timestamping of incoming packets is done at the hardware level, the MAC layer. We build our system using OS X and iOS, which does not provide an API to get the timestamps directly from the hardware layer, and this makes many of the protocols a bad fit for our implementation. The lower precision of the timestamps we can get would lead to an imprecise synchronization protocol. We need to base our implementation on a protocol where the effects of an imprecise timestamp has the least effect on the synchronization precision; we need a protocol with the shortest possible critical path.



(a) Critical path using standard unicasting between nodes. Worst-case is 710 milliseconds

(b) Critical path is much smaller with reference broadcasting. Worst-case is 110 milliseconds

Figure 4.1: The critical paths in two different message sending scenarios.

Many of the synchronization protocols unicast messages between nodes creating a large critical path. Unicasting messages require that each message pass through all four stages of a packet delay; send(1), access(2), propagation(3) and receive(4). Broadcasting, on the other hand, only need to go through stages one and two once, and when the message is on the network each receiver only goes through stages three and four. This helps dramatically to decrease the critical path as we cut out a potential delay of up to 600 milliseconds; the remaining worst-case delay is 110 milliseconds. Figure 4.1 shows the difference between these two casting methods.

The reference broadcast synchronization protocol (RBS), by Elson et al. [3], use broadcast messages in their algorithm to achieve a short critical path. Besides shortening the critical path the authors of RBS found that the third stage, the propagation time, of a packet delay can be removed. They exploited that broadcast messages will arrive at all receivers almost instantaneously, the actual travel time being defined by the laws of physics. If you are broadcasting on Wi-Fi the travel time will be defined by the speed of radio waves through air, and if you are using Ethernet cables it will be the speed of electricity through cable. Both cases are almost as fast as the speed of light, which is roughly 300000 kilometers per second. This means that the messages will arrive very quickly at the receivers unless your router is extremely far from your receivers. This, which we try to verify during our evaluation, means it is possible to create a synchronization algorithm that is solely based on the fourth stage of a packet delay; the receive time. Because of the short critical path, and the negligible propagation time, we base our algorithm on the RBS protocol.

While RBS has several benefits, it also has a few things we don't like, such as a high number of message exchanges. In a system with n nodes, RBS uses $O(n^2)$ message exchanges for each synchronization pulse. This is because each node unicasts their observations to the other nodes in the system, we wish to improve on this by having the nodes broadcast their observations instead. By broadcasting the observations each node only needs to send one message for each synchronization pulse it observes, cutting the number of message exchanges to $O(n)$.

RBS requires a special root node for sending pulse broadcasts, but we don't want different types of nodes we wish that every node in the system behave exactly the same. We were inspired by the Reachback Firefly Algorithm (RFA) [16] where every node in

the system both sends and receives pulses. Therefore, we added this behavior by distributing the broadcast pulses onto every node. After we moved the responsibility of pulse broadcasts onto every node there were no longer any differences between the nodes. This makes the system easier to use and understand since there are no special cases to consider; you run the application on the number of devices you wish and it should just work. This structure can be thought of as a distributed master / slave setup, where each node is both a master and a slave at the same time.

Because we use broadcasting of pulse references, the system requires a minimum of three nodes before it can synchronize the nodes. This is because a node cannot use its own pulse broadcasts as a reference. We use the fact that once a message is on the network the message will be received almost simultaneously by all nodes. However, when a sender of a broadcast listens for its own broadcast messages these are sent back to the sender via a loop-back in the network stack. Therefore, the sender will receive the message from another medium than the rest of the nodes, thus making our assumptions about simultaneous reception false. We have done testing where we tried to disable the loop-back function in the `IP_LAYER` hoping that this would force the message to go onto the network before being returned to the sender. Nevertheless, disabling the loop-back did not have that effect, instead it just disabled the senders' reception of the message.

We could have created an algorithm with a lower bound of two nodes by switching to a remote-clock reading approach instead, but this would lead to a less precise synchronization. For the algorithm to work with only two devices, with the tight synchronization, would require that the network kernel of OS X and iOS would allow broadcast or multicast messages to be returned to the sender from the network and not by direct loop-back.

Algorithm Description

Our algorithm does three things; it sends pulse broadcasts, it receives pulse broadcasts, and it calculates the local offset to incoming command messages. A pulse broadcast is a UDP message containing a unique id, which is used when the nodes exchange observations about when they received a specific pulse. The pulse is sent to all nodes in the system using UDP broadcasting. When a node receives such a pulse message it records its local time and sends this information as a new UDP broadcast to the other nodes.

A high level description of our algorithm can be done in three steps (more details of this process can be seen in Figure 4.4 on page 23):

1. A node transmits a pulse with an id to all connected nodes.
2. Each receiver records the time that the pulse was received according to its local uptime.
3. All receivers exchange information about their observations.

In step two, each receiver records the reception time according to its local uptime. Uptime is provided by the local system by calling `mach_absolute_time()`, a low-level call which is the finest-grained timepiece available on OS X and iOS. Uptime is the time elapsed since the last boot of the device and it is not affected by clock adjustments done manually or automatically by the systems' implementation of NTP. Using uptime, instead of the systems clock, prevents errors induced by clock adjustments because we are using

an ever-increasing reference. Our algorithm depends on the fact that the time on each node only runs forward, if it were to run backwards our calculations would no longer be precise.

Commands

To control the nodes in our system we have created playback commands. These commands are triggered by pressing the corresponding buttons in the user interface. A node can play, stop, fast-forward, reverse or choose a different output. When a command is triggered on a node, it will send a TCP message containing that command to all the other nodes. The receiving nodes will then calculate their own local offset compared to the senders' with our linear regression calculation.

Linear regression

To synchronize the nodes in our system we use simple linear regression, which allows us to model the relationship between the observed timestamps. Simple linear regression fits a straight line through the set of observed timestamps such that the distance between the observation points and the fitted line is as small as possible, see an example in Figure 4.2.

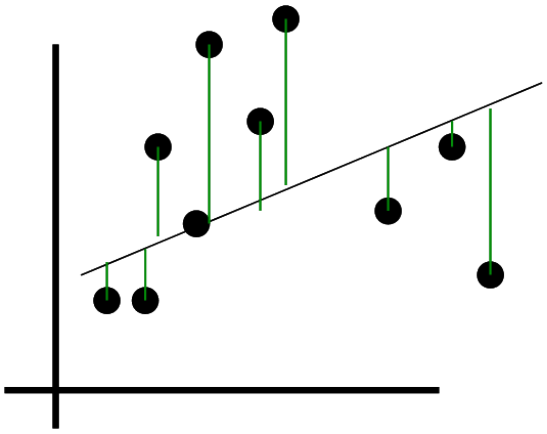


Figure 4.2: Example of simple linear regression. The black dots are the observed values, the straight line is the best fit and the green bars highlights the difference between the observed values and the best-fit line.

When a node triggers a playback command, this command is sent to all the other nodes. The playback command contains a timestamp from the sender of when the playback command will take place locally at the sender. Since all nodes have exchanged pulse observation times, they can then use this data as input to the linear regression function. So that when a sender sends a command message to a receiver, the receiver can construct the input data needed for the function.

The input is a list of tuples; one for each observed pulse that has been observed by both the sender and the receiver. Each tuple contains two timestamps; the time of observation from the sender and the receiver. From these historical observation points, the linear regression function can predict what the local time will be at the receiver when

the sender will execute the playback command. Our calculations need at least two pulse observations before it can predict accurate timing information.

With the help of statistics, we can calculate the precision of our prediction. We need two values to do that; the total sum of squares (SST) and the sum of squared residuals (SSR). SST is the sum of the differences between the observed data points and the average of all the data points where each difference, also called the residual, is squared. SSR is the sum of the differences between the observed data points and the best-fit line, again all the differences are squared. The precision of the linear function can be found by dividing the SSR with the SST, this is also called the coefficient of determination or r^2. This is calculated every time a node receives a command and our precision or r^2 is consistently around 99% in the tests we have done.

Missing observation data is handled gracefully by the algorithm as seen in Figure 4.3 where the best fit line is calculated even though some observation points are missing.

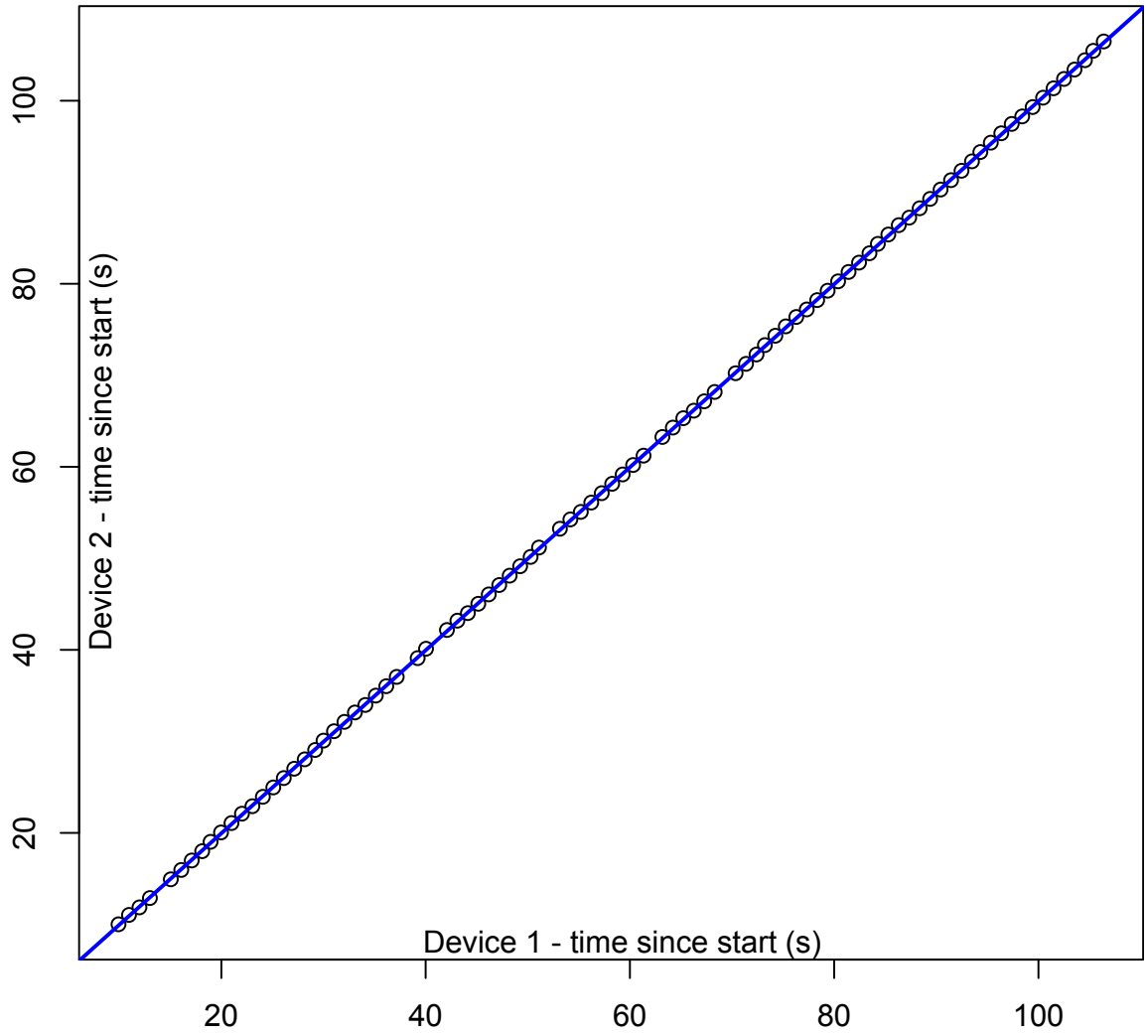
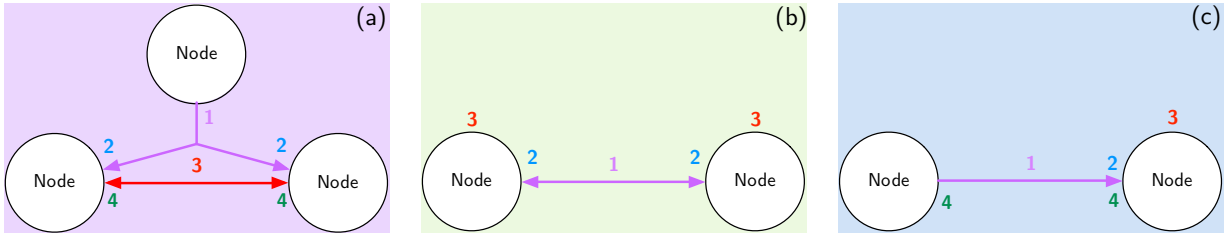


Figure 4.3: The circles show when device 1 and 2 received a pulse relative to the elapsed time since starting the application on the device. The blue line is the best-fit line for predicting timestamps. Missing circles indicates a pulse that was not received by both devices.

4.2.2 Communication

Communication is an essential part of any distributed application since such applications does not share resources like memory or storage. All work towards a common goal in a distributed system must be accomplished through message exchanges. In our system, the common goal is to make all nodes in the system generate output at the same time. We accomplish this with three communication flows. First, the broadcasting of the pulse references. Second, the discovery protocol, which builds and maintains the network of connected nodes. Third, the sending of commands from one node to all other connected nodes. The three types of communication flows is illustrated in Figure 4.4.



1. Broadcast a PulseId
2. Note ObservationTime of PulseId
3. Broadcast ObservationTime of PulseId & NodeId
4. Note received observation time of PulseId & NodeId

1. Broadcast Discovery & StartTime & NodeId
2. Update LastSeen recording of NodeId
3. If a nodes StartTime changes it has restarted, then remove all recorded Pulse observations for that NodeId

1. Send Command & ExecutionTime & NodeId
2. If enough Pulse observations from NodeId are available go to 3
3. Transform Commands ExecutionTime to LocalTime with linear regression
4. Sender performs Command at ExecutionTime, receiver at LocalTime

Figure 4.4: Three communication flows from our solution, **(a)** Pulse Broadcasts (UDP), **(b)** Discovery Protocol (UDP), and **(c)** Command Protocol (TCP).

Discovery and Pulse References

To minimize the number of message exchanges we have combined the discovery protocol with the broadcasting of pulse references. Nodes repeatedly sends a pulse reference as a UDP broadcast at a random interval anywhere between zero and two seconds. The reason for the random interval is to avoid any message clotting that could arise if all nodes sent their broadcasts at the exact same time. Because the intervals are random, there should be less risk of this happening.

The pulse message is a triple containing a unique id for the pulse along with the id and start time of the sender. The pulse and sender ids are used to build the input data in the linear regression calculation. The senders' id and start time are used by the discovery protocol.

When a node receives a pulse message, it will start by creating a timestamp. This timestamp is added to the local pulse observations along with the pulse id. It will then send a new broadcast message, a pulse observation triple containing the pulse id, its own id and the time of observation. When a node receives such a pulse observation, this information is added to its local data structure for later retrieval by the linear regression function.

After the pulse observation is sent the discovery protocol starts to process the pulse message. First, a last seen timestamp is created which is used to determine when a node goes missing. If this value is not updated for 60 seconds we assume that the sender has disconnected from the system. The protocol uses the senders' id as key in a map for a tuple containing the senders' start time and last seen timestamp. If the senders' start time changes, because it has restarted, the previous pulse observations from this node can no longer be used and they are removed.

The discovery protocol also keeps updating a map where the senders' id is the key and the senders' contact information is the value. This information is used by the command flow.

Command Flow

The commands are sent using TCP communication because we rely on commands reaching their destination. TCP uses acknowledgment replies for each message sent and automatically resends messages, which fail to be delivered. UDP communication does not try to resend messages and therefore has a lower certainty of reaching its destination. If a command does not reach its intended target, that node will not behave according to the group behavior and this is highly undesirable.

When a node triggers a command, it will be unicasted to all the connected nodes in the system.

4.2.3 Redundancy

The biggest problem in traditional master / slave setups is that they have a single point of failure. Typically, you have one device generating time code, the master, and any number of devices following that time code, the slaves. However, if the master stops sending time code the slaves will also stop following along.

For our solution we needed an approach without a single point of failure. What we did was to distribute the responsibility of sending synchronization references and playback commands onto every node in the system. A strategy that can be thought of as a distributed master / slave setup where every node is a master and a slave at the same time. Every node in our system can send playback commands which will then be executed simultaneously on all connected nodes.

Our solution does not have a single point of failure, if a node drops out the remaining nodes will be unaffected.

4.2.4 Output

Our system currently has two types of output; MIDI Time Code (MTC) and audio playback. Both types are kept synchronized across all nodes in the system.

MIDI Time Code

MTC is a sub-protocol of MIDI that is used to synchronize devices that perform timed performances. MTC is a MIDI adaption of the Society of Motion Picture and Television Engineers (SMPTE) time code. SMPTE timing information consist of hours, minutes, seconds, and frames.

Several MIDI messages make up the MTC protocol, but the most important message is the quarter frame message. MTC breaks SMTPE timing information into eight 4-bit quarter frame messages as seen in Figure 4.5. Quarter frame messages are sent at a rate of four per SMPTE Frame, and all eight frames must reach their destination before the receiver can decode it into full timing information again. Because timing information is sent in eight messages with four sent per SMPTE frame, the time increments on the clients happen every two SMPTE frames.

Message	Value
1	Current Frames Low Nibble
2	Current Frames High Nibble
3	Current Seconds Low Nibble
4	Current Seconds High Nibble
5	Current Minutes Low Nibble
6	Current Minutes High Nibble
7	Current Hours Low Nibble
8	Current Hours High Nibble and SMPTE Type

Figure 4.5: The eight 4-bit quarter frame messages used to generate MTC. The first message contains the low nibble (bits 0 to 3) of the Frame. The second Quarter Frame message contains the high nibble (bits 4 to 7) of the Frame. The third and fourth messages contain the low and high nibbles of the Seconds. The fifth and sixth messages contain the low and high nibbles of the Minutes. The seventh and eighth messages contain the low and high nibbles of the Hours. The eighth message also contains the SMPTE frames per second Type (24, 25, 30 drop, or 30 fps).

We have built a MTC generator using the CoreMIDI API exposed by Apple. This API allows creating MIDI endpoints and using these for sending or receiving data. Our generator has one endpoint, a MIDI out port, used for sending our time code. Other applications running on the same device will see this port, when our application is running, and can choose to listen in on the data coming from this port.

MTC needs to be very precise, meaning that the interval between quarter frames must be correct and constant. If we run MTC at 25 frames per second we need to generate four quarter frame messages per frame, 100 messages per second. If the interval varies, the time code becomes less precise and will cause errors for anyone using it as a time source. To keep our time code precise we spent some time trying to optimize the precision of our

generation algorithm. Initially we built each quarter frame just before it was to be sent; at that point, we had a precision of around six milliseconds. We switched to building all eight quarter frames before the first was sent and our precision has been improved to 0.842 milliseconds.

Audio Playback

Alongside the ability to generate MTC, our solution is capable of playing back audio files that are kept coordinated throughout all connected nodes. This makes it easy to demonstrate and test the application without having to setup a third party application that understands MTC. Our playback is based on the AVAudioPlayer API, which provides asynchronous playback capabilities.

The API allows preloading of the audio players buffer, which should minimize the lag between calling the play function and the playback actually starting. However, we did experience problems with this because it seems to have a few bugs in these preparation functions. The lag between calling play and the audio playback actually starting varies quite a lot, from 20 milliseconds up to around 500 milliseconds. After researching these problems, we found that other users experienced the same problems and took various approaches to get around it.

For us the problem is that we want to synchronize the playback across several machines, but if the actual playback lag, is as high as 500 milliseconds this becomes hard to achieve. This uncertainty is by far the largest non-deterministic delay in our system; the synchronization lag is small compared to this delay in the audio player.

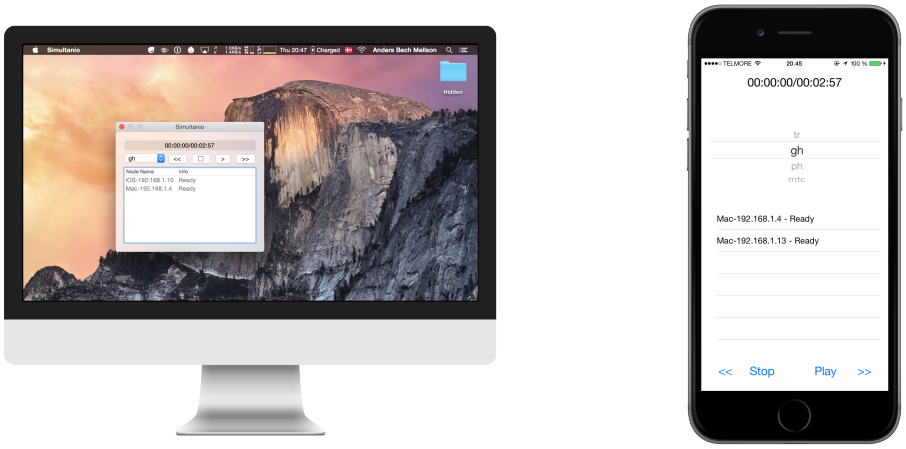
Our solution is to load an audio file containing silence into the player at node startup and play this 10 times. Each time we measure how long it took before the playback started and we use these measurements to create an estimated lag introduced by the player. When a node receives a command, it will offset the synchronization calculations with the estimated lag of the player. Far from being ideal this has been an acceptable solution thus far, creating a much smaller lag around two milliseconds.

4.2.5 User Interface

The user interface of our system is very minimal. We have a menu for choosing the desired output, and in the current version, this is a choice between a few musical pieces and the MTC generator. Then we have the playback controls for rewinding, stopping, playing and fast-forwarding. We use a display for showing local node information and timing information during playback. Last, we have a display showing information about all the connected nodes in the system. This display updates every second and reflects the synchronization status of the nodes and it will show when nodes drops out. Figure 4.6 shows how the user interface looks running on OS X and iOS.

4.3 Summary

In this chapter, we went through our technology exploration phase which ended up with a choice to use Swift over Xamarin. We have looked at how we implemented the moving



(a) OS X version running on iMac.

(b) iOS version running on iPhone 6.

Figure 4.6: The user interfaces of Simultanio.

parts of our system, namely communication, synchronization and output generation. A software architecture of our solution can be found on page 50.

Chapter 5

Related Work

In this chapter, we will look at how others have tried to solve the problem of synchronizing multiple systems on a commercial operating system using the network stack. We will present a select few of the solutions available today and how they compare to the solution we envision.

One way of creating a solution is to use ready-made building blocks. The company Cycling 74 has created a modular software called Max that provide building blocks for musical tools and instruments. The third expert we interviewed has created a Max patch that can synchronize several systems. It is a master / slave setup where one device is outputting sync messages, 480 times per beat, to all the slaves that are listening. The setup works using wired Ethernet cables on a closed network setup. Each message is unicasted to a slave with a known IP address. Because of the high number of sync messages, the system achieves a tight synchronization. This enables a musical approach where the master can dictate the tempo along with the measure. There is no dependency of the clock in the slave because the master triggers every beat. The system can synchronize at least eight devices simultaneously. The downside to this solution is that it is not resistant to node failures because it is a master / slave setup. Another problem is that the master needs to create and send 480 messages per beat per slave. This can quickly create a large amount of outgoing messages from the master. Moreover, while this seems to work satisfactory on Ethernet connections, our expert has had little success using the Max patch on Wi-Fi connections.

Instead of using ready-made software building blocks, you can also build a solution with a general-purpose programming language. The company Sononum has done that and created a time code generator for OS X called Horae[1]. Horae can generate MIDI Time Code (MTC) and Linear Time Code (LTC) on OS X. This has historically been done on expensive hardware units called master clocks. The application outputs accurate time code with a minimal amount of jitter, see just how accurate in our evaluation on page 30. Horae is an impressive technical achievement, the time code generation is flawless. The only downside is that it is not a distributed solution; Horae is a single machine application. In addition, while it is possible to distribute the generated time code, putting it in the master / slave category, it will still create a single point of failure.

[1] `https://sononum.net/horae`

The Digital Audio Workstation (DAW) Apple Logic Pro[2] can also generate MTC, which can synchronize many slaves capable of reading MTC. Logic can be used as both a master and a slave because it can both read and generate MTC. Logics solution is also a non-distributed master / slave setup mirroring all the regular problems of this domain.

The Dutch company Showsync[3] provides several solutions for synchronization; in the form of plug-ins for Ableton Live. Their plug-ins are built using the Max platform and is tightly tied in with Ableton Live through a product called Max for Live. Showsync extends the capabilities of Ableton Live with MTC generation and synchronization of internal parameters, such as volume, mute, pan and playback of clips. The plug-ins are an easy way to achieve synchronization if you are already running Ableton Live, but the precision of the MTC generation is not good as we will see in our evaluation. Moreover, all their plug-ins are master / slave based inheriting all the problems from this domain.

Avid has created a distributed synchronization solution for their own video- and audio software called Satellite[4]. Compared to the other solutions Satellite has redundancy built in, so that if a node drops out the others will keep playing. Satellite works on specific network hardware and only using Ethernet connections. The maximum number of devices that Satellite can synchronize is 12 audio devices, slightly less if video devices are part in the system. This solution is close to the envisioned goal of this thesis, with one major difference; it is a closed system because it only works with software from Avid. Avid Satellite is an attractive technical solution, and we will use this as inspiration to create a more open system. We will also aim for a solution capable of synchronizing more than the 12 devices achieved with Satellite.

5.1 Summary

In this chapter, we have looked at some possible ways to synchronize several devices in a stage environment. Expert three from our expert panel has created his own solution with the building blocks provided by Max from Cycling 74. Showsync also use Max to create plug-ins that can synchronize Ableton Live running on multiple devices. Horae and Apple Logic Pro can create stable time code, which a few years back was done mainly by hardware units. Avid Satellite is a modern approach, building on the foundations of distributed computing. We will use both solutions as an inspiration in our own work.

[2] `http://apple.com/logic-pro/`
[3] `http://showsync.info/`
[4] `http://avid.com/US/products/Satellite-Link`

Chapter 6

Evaluation

In this chapter, we will evaluate the software we have produced throughout this project. Besides these software evaluations, we will also evaluate network propagation properties on standard routers and socket response time on OS X.

First, we will evaluate the MTC generation module of our solution against three other software MTC generators. Second, we will look at the propagation properties on a home network, verifying an essential claim in one of our referenced papers. Third, we evaluate how much faster the kernel space is in receiving network messages compared to the user space. Fourth, we verify that our modifications of the library CocoaAsyncSocket was beneficial. Finally, we do a full system evaluation of our main contribution, the distributed application Simultanio, against the requirements.

6.1 MTC Precision

As we talked about in Section 4.2.4 the precision of MTC is essential when it is used as a time source. In this section, we will evaluate how precise four different MTC software generators are. We evaluate the preciseness of generated MTC by measuring the interval between incoming MTC quarter frames. When MTC is running at 25 frames per second it is generating 100 messages per second, a message every 10 milliseconds. Therefore, we know that the interval between each received MTC quarter frame should be exactly 10 milliseconds, hence we can use this knowledge to measure how far from this interval each generator is. The application we have created, for testing the MTC preciseness, looks at this interval and from this calculates calculates how far the incoming time code is from the ideal.

Evaluation Setup

For this evaluation, we used one Mac running OS X 10.10. The device was running our MTC preciseness testing application, Time Code Tester[1]. It was also running four different time code generators; Horae[2], Logic Pro X[3], Showsync LiveMTC[4] and Simultanio.

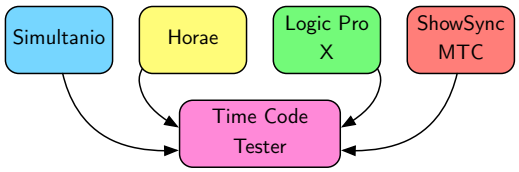
[1] `https://github.com/mofus/Time-Code-Tester`
[2] `https://sononum.net/horae`
[3] `https://www.apple.com/logic-pro`
[4] `http://showsync.info/livemtc`

Evaluation Result

We evaluated each time code generator in separate test runs, testing with only one application running at a time. The generators were set to generate MTC at 25 frames per second and have it send that time code to our Time Code Tester. We left the generators running for five minutes each and collected data about mean precision and worst-case precision. The mean precision was calculated as the mean of all the data points, while the worst-case was the largest error observed during the five minute run.

The data from the test is shown in Figure 6.1 which shows that our application is third best. The most precise generator is the application Horae that achieve remarkable results. Horae is built with Objective-C and C++ and our application is built using Swift. Michael Tyson, the creator of Loopy and Audiobus[5], has done some testing on the real-time performance of Objective-C vs Swift[6] and found that Swift is not as precise as Objective-C. His findings can help explain some of the reason why our application is less precise than Horae and Logic Pro X. Even though we are using Swift, we still think that better performance can be achieved by spending more time trying to optimize the generation algorithm.

However, our application performs much better than the generator from Showsync and their application has been used to good effort at big festivals such as the Dekmantel Festivals main-stage, in front of many thousand spectators. We set out to reach a solution, which is "good enough" and if the Showsync application is good enough for big festivals our application fulfills this goal since we get even better precision.



(a) Setup showing applications sending MIDI time code to our testing application.

	Mean Error	Max Error
Horae	0.036 ms	0.199 ms
Logic Pro X	0.079 ms	0.321 ms
Simultanio	0.842 ms	5.231 ms
Showsync MTC	1.770 ms	19.255 ms

(b) Results from running each of the generators for five minutes.

Figure 6.1: Setup and results from the MTC preciseness evaluation.

6.2 Propagation Time

This evaluation will, hopefully, verify the claims made by our reference seminal paper on synchronization using broadcast pulses [3]. The claim is that the propagation time for a message on a small network is extremely low. In fact, the delivery should be so fast that a broadcast message will arrive at all receivers at almost the same time. The main contributor to the delay time is the laws of physics, if using Wi-Fi this would be the time it takes a radio signal to travel from the router to the receiver and when using Ethernet it would be the time, it takes the electricity to travel through the cable.

[5] `http://atastypixel.com/`
[6] `http://masterpieceedition.tumblr.com/post/104890380933/objective-c-or-swift`

To verify this we have devised a test looking at intervals between reception times of an oscillating pulse signal. This signal is created by broadcasting a message, to all listeners on the subnet, at a fixed interval of one second. If the broadcast messages truly arrives at every listener at the same time, then the intervals between reception times should be the same.

We performed this evaluation for both multicasting and broadcasting to see if one method had better properties than the other did. The evaluation was also conducted on two different routers, to eliminate hardware skew.

6.2.1 Evaluation Setup

For this evaluation, we used three Macs running OS X 10.10 and three iOS devices running iOS 8.3. All devices were running our main application, Simultanio.

6.2.2 Evaluation Result

This evaluation shows that Macs receive broadcast and multicast messages on average within 0.077 milliseconds from each other. The data shows that iOS devices are much less precise; on average, they receive messages within 6.630 milliseconds from each other.

This result is skewed by the fact that we do not have a precise timestamp on the incoming packets. Our thoughts are that if we had the capability to timestamp at the hardware level we would see that the differences would be much smaller. So while we cannot completely verify the claims made by Elson et al., we trust their findings. The next couple of evaluations will look at how close we can get to hardware timestamps with the Apple devices we are using.

	Broadcast Mean Error	Broadcast Max Error	Multicast Mean Error	Multicast Max Error
Mac router 1	0.096 ms	24.105 ms	0.078 ms	27.238 ms
Mac router 2	0.070 ms	228.947 ms	0.065 ms	78.367 ms
iOS router 1	8.830 ms	63.636 ms	8.105 ms	62.227 ms
iOS router 2	3.963 ms	220.497 ms	5.624 ms	78.291 ms

Figure 6.2: Results from the propagation time evaluation.

6.3 Kernel Space Timestamps

In the search for a more precise timestamping method, we contacted Apple through their developer forums. We wanted to know how to get timestamps as close to the hardware as possible. An Apple engineer suggested that we use the Berkeley sockets API that allows getting the timestamp when the message arrives at the kernel level of the OS, before it is passed to the user level. To verify that this would improve preciseness we did an evaluation.

This is an evaluation of the time it takes for the kernel to send a message to the user space using interprocess communication since OS X and iOS is based on the Mach kernel.

6.3.1 Evaluation Setup

For this evaluation, we used two Macs running OS X 10.10. The first Mac was used for transmitting network messages to the second Mac. The second Mac was used to timestamp incoming network packages, first in the kernel space and secondly in the user space. Both devices are running applications originally created by Max Vilimpoc[7], we have modified his code slightly to make it compile and run on OS X.

6.3.2 Evaluation Result

The sending device sent a message every second to the receiving device. After 500 readings, the average difference between the kernel and user level timestamps was 52.81 μs. This equals 0.05 milliseconds and hence does not seem the right place to go looking for a massive improvement to the synchronization algorithm.

6.4 CocoaAsyncSocket Modification

This evaluation is meant to evaluate our modification to the CocoaAsyncSocket library. The library calls a delegate method after it has received a package. Our modification adds a timestamp before calling the delegate. This evaluation will measure the difference in time from receiving the message to the delegate handler being called.

6.4.1 Evaluation Setup

For this evaluation, we used one Mac running OS X 10.10 and one iOS devices running iOS 8.3. All devices are running our main application, Simultanio. We compared the timestamps we provided in our modification of CocoaAsyncSocket with a timestamp created when the delegate was called.

6.4.2 Evaluation Result

After 500 reference pulses our evaluation showed that our added timestamp was 280.37 μs before the delegate timestamps. This equals 0.28 milliseconds, and while it is not a big improvement, it does make our algorithm slightly more precise. We conclude that our added timestamp improves the ability to use this library in time sensitive applications such as our own.

[7] `https://vilimpoc.org/research/ku-latency/`

6.5 System

To ensure that our system fulfills the requirements identified earlier, we have conducted a series of evaluations. As described in Chapter 3 our identified requirements are:

1. Support OS X targeting Apple laptops computers.
2. Support iOS targeting Apple hand-held devices.
3. Node dropouts must not affect playback of other nodes.
4. Easy, reliable and reproducible setup procedure.
5. Synchronization tightness of 30 milliseconds or below.

Requirements 1 and 2 are fulfilled since our solution is running on OS X and iOS. Requirement 4 is also fulfilled since the solution automatically discovers and handles nodes running on the same subnet.

To verify that our solution meet requirement 3 we did a redundancy evaluation, which follows this.

Finally, to verify that our solution fulfills requirement 5 we did several hours of evaluation in various networking conditions. That evaluation will round of this chapter on evaluation.

6.5.1 User Interface

We set out to create a user interface conveying playback, synchronization and redundancy information that a stage performer would find useful. The current version of the user interface, as seen in Figure 6.3, have been shown to two experts which deemed it satisfying. Both experts wanted more information displayed, such as the ability to set and see playback markers. We agree with the experts that the current interface is very limited and needs further work.

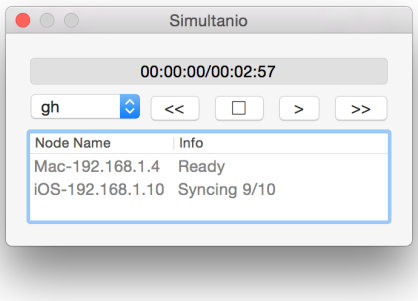


Figure 6.3: Screenshot of Simultanio running on OS X.

6.5.2 Redundancy

In this evaluation we wish to verify that our solution is tolerant to node dropouts.

Evaluation Setup

For this evaluation, we used two Macs running OS X 10.10 and two iOS devices running iOS 8.3. All devices are running our application, Simultanio. The nodes was used to playback an audio file in sync with the other nodes.

Evaluation Result

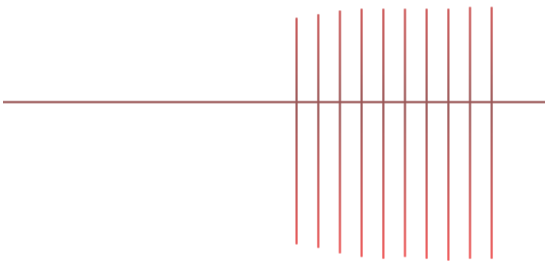
We started playback from one node, which was reflected by the other three nodes that also started to play. After the system had been running for one minute we switched off the node from which we started the playback. The remaining nodes was not affected and kept playing. We repeated this way until only one node was playing. This verifies that our system can handle node dropouts.

6.5.3 Synchronization

Evaluation Setup

For this evaluation, we used five Macs running OS X 10.10 and four iOS devices running iOS 8.3. All devices, but one, are running our application, Simultanio. The last device is running Ableton Live to record the audio playback of the other nodes. The devices are playing audio files, which we used in our measurements.

Each node played a click sound every second and we recorded these clicks into a Digital Audio Workstation (DAW), Ableton Live 9. Post recording we analyzed the audio files using the software `aubio`[8] that includes a tool, which can do transient analysis of audio files and output the results to a text file. These text files contain the relative position of a detected peak in the audio file.



(a) Waveform showing 10 clicks.

Click 1: 13.579751 s
Click 2: 14.578367 s
Click 3: 15.582222 s
Click 4: 16.580929 s
Click 5: 17.579388 s
Click 6: 18.578321 s
Click 7: 19.582018 s
Click 8: 20.580566 s
Click 9: 21.579048 s
Click 10: 22.580521 s

(b) Analysis showing the same clicks.

Figure 6.4: Audio file analysis of example recording with 10 clicks.

[8] `http://aubio.org/`

The textual representation of the audio file transients was processed with a custom shell script, which combines the analysis results from all eight recordings into a single file. The combined file was then loaded into a spreadsheet for further analysis.

This evaluation was executed in two different scenarios, each under two varying network loads resulting in four different tests:

- Scenario 1

    1. Low Traffic
       15 minutes recording where all nodes are triggered to play for 10 seconds every 15 seconds. This is done under low traffic network load, where the main traffic is generated by the synchronization protocol.
    2. High Traffic
       15 minutes recording where all nodes are triggered to play for 10 seconds every 15 seconds. This is done under high traffic network load, where a node sending UDP broadcast messages every 10 milliseconds generates the main traffic.

- Scenario 2

    3. Low Traffic
       1 hour recording where all nodes are triggered to play once and then allowed to drift apart. This is done in a low traffic network similar to test 1.
    4. High Traffic
       1 hour recording where all nodes are triggered to play once and then allowed to drift apart. This is done in a high traffic network similar to test 2.

Recording Equipment

One Mac running OS X 10.10 recorded the eight nodes using the DAW Ableton Live 9. The recordings were done with a Behringer Ultragain using optical cabling to a RME BabyFace USB sound card. The eight nodes were connected to the Behringer Ultragain using analog mini jack to jack cables. The audio levels was manually calibrated before running the evaluations.

Figure 6.5: Evaluation setup.

	Mac Mean Error	Mac Max Error	iOS Mean Error	iOS Max Error
Test 1 low traffic	1.380 ms	81.269 ms	15.274 ms	63.599 ms
Test 1 high traffic	1.719 ms	92.896 ms	6.501 ms	39.596 ms
Test 2 low traffic	6.622 ms	46.631 ms	15.410 ms	56.871 ms
Test 2 high traffic	6.958 ms	52.247 ms	8.542 ms	34.913 ms
Average	4.170 ms	68.261 ms	11.432 ms	48.745 ms

Figure 6.6: Results showing mean deviation along with the maximum offsets.

Evaluation Result

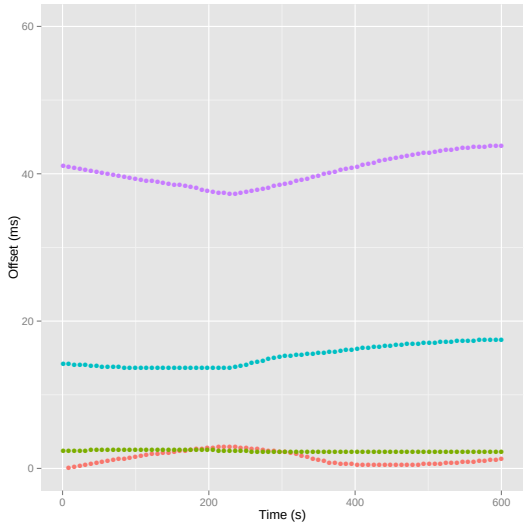
The summarized results of the evaluation can be seen in Figure 6.6. The table shows the mean error and the maximum error, the mean error, also called mean deviation, is the offset to the mean of all results and the maximum error is the largest offset between any two nodes observed. Row 1 shows the results from test 1 under low network traffic. These show that the Macs achieves a mean precision of 1.380 milliseconds and that the maximum observed error was 81.269 milliseconds. The maximum error detected could be interpreted as an out lier as the result was only observed once, if that observation was removed the Macs maximum error would be 5.799 milliseconds instead. The same is true for test 1 under high network traffic where the worst offset for the Macs is 92.896 milliseconds, but again this is only observed once and if removed the maximum error would drop to 5.859 milliseconds. The reason for these outliers are probably caused by the non-deterministic amount of time it takes OS X to start playing the click sound we use to measure the offsets.

The entire result sets are shown in Figure 6.7 and to help emphasize the trend lines the data has been cleaned using `R`'s[9] smoothness estimation algorithm[10] from the package `ggplot2`[11]. Before this smoothing, the graphs were too unclear to convey any clear trend lines. After smoothing, there are some clear trends in all four evaluation runs. Test 1 shows that the Macs are very precise and do not drift, the iOS devices are less precise and drifts slightly. What we did not expect was to see that the iOS devices perform better under high network load. Our initial reaction was that this could be because the are less likely to do any power saving when the messages keeps coming in.

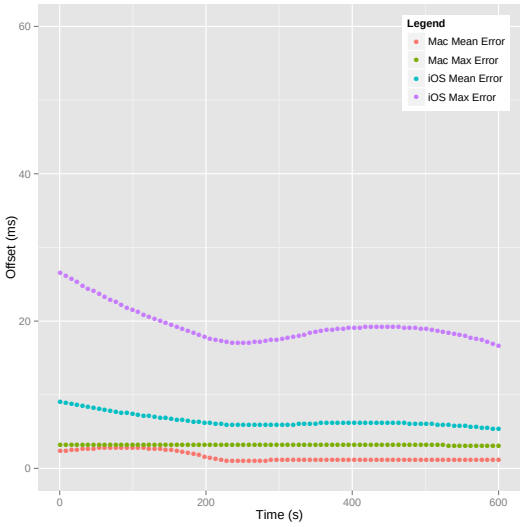
Test 2 shows that all the devices drift apart when they are left unsynchronized, in the worst-cases the drift is around 50 milliseconds. If these trends continues, there would be a drop of precision by 1200 milliseconds every 24 hours. In this test, the iOS and Mac devices are more equally matched, but the Macs still have a slightly better average precision.

According to the studies by Gurevich et al. the distance between performers must be below 10 meters, similar to 30 milliseconds, if the performers are to keep the same tempo and thereby play coherently [4]. Our solution achieves mean errors way below this threshold and therefore we think that our solution is able to solve the problem of synchronizing humans and computers on stage.

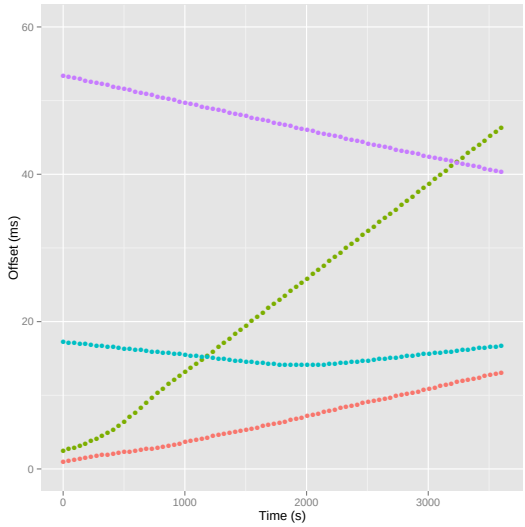
[9] `http://www.r-project.org/`
[10] `http://www.inside-r.org/r-doc/mgcv/gam`
[11] `http://ggplot2.org/`



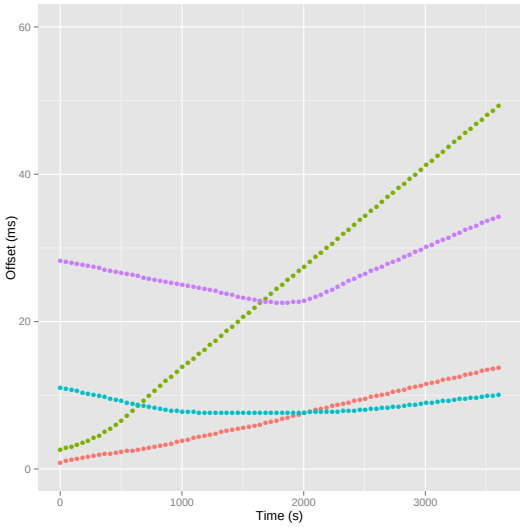
(a) Test 1 low traffic.



(b) Test 1 high traffic.



(c) Test 2 low traffic.



(d) Test 2 high traffic.

Figure 6.7: Smoothed results showing mean deviation along with the maximum offsets. Graph (a) and (b) both show data from 15 minutes where (c) and (d) show data from 1 hour. Flat lines, as close to zero as possible, is the ideal.

6.5.4 Experts Opinion

We had our first expert see and try our system, at a hands on demonstration, at the expert's office in Copenhagen. The expert was impressed by the solution and feels that it can solve many of the problems with synchronization on stage.

During startup of our system, nodes are automatically discovered and added to the number of nodes that form the distributed application. This was deemed a transparent and helpful way to perform the setup phase of a synchronization system. The status information displayed, about connected nodes in the user interface, was found to be instructive and usable. Most importantly, this expert found that the tightness of the synchronization was very good and indeed able to solve synchronization problems.

Our second and third expert evaluated the system by watching a video recording of the system in action. Expert two has been unavailable for a comment before our deadline but expert three did provide some positive feedback. He finds our solution elegant and especially the fact that it breaks away from the traditional master / slave setup is highly praised. The tightness of our synchronization is found to be on par with the experts own solution which is in use today.

6.6 Summary

In this chapter, we did evaluations of all the moving parts in our solution along with a few network properties. The evaluation of our main contribution, our distributed application Simultanio, showed that we can synchronize OS X nodes to a precision of 4.17 milliseconds, iOS nodes have a lower precision at 11.43 milliseconds. Our measurements were done at the audio output of all devices, as we wanted a realistic reflection of how tight the nodes can play synchronized audio.

Besides evaluating our own work we also tried to verify the claims on propagation delays as put forward in the work by Elson et al. [3]. Our evaluation verified that messages arrive very close to each other, but we could not verify it completely because of the imprecise timestamping methods available on OS X.

Chapter 7

Discussion

In this chapter, we will reflect on our project. First, we look at our results from the evaluation and what they imply. Second, the lifespan of our solution is taken into consideration. Third, we will share our thoughts on our final solution - do we actually solve a problem? Lastly, we look at future work for the system.

7.1 Evaluation Results

7.1.1 Measurement Uncertainties

There is uncertainty in our measured results because we do a full stack evaluation, meaning that our measurement point is at the audio output of the devices. Measuring at this point creates uncertainty because the time it takes from we call the audio API until there is actual output is non-deterministic, we have observed the delay to vary from 20 to 500 milliseconds. If we were to only measure the preciseness of our synchronization algorithm, we would ideally measure at the point before triggering the audio output. In our main reference paper [3] they test their precision using a specially developed Linux kernel module that has virtually no uncertainty. If we used a similar technique, we would probably achieve much better results. In our case, we have done the measurements with the audio APIs that Apple provide because our initial end users work in the audio world. We have also tried to minimize the uncertainty of the audio API by calibrating it at node startup. Nevertheless, there is still a non-deterministic delay that we cannot control and this is not ideal.

7.1.2 Humans vs Computers

Our evaluation showed that we could synchronize computing devices to be within 11 milliseconds from each other. We compared this to the research by Gurevich et al. [4], which shows performers must be within 30 milliseconds from each other to keep a stable tempo while playing. Our assumption is that if humans can synchronize their actions to other humans with a maximum delay of 30 milliseconds, then surely they must be able to synchronize their actions to a computing device with a maximum delay of 11 milliseconds. However, the scenarios are different. Humans playing with humans have the capability of adjusting their actions on both ends. When humans play alongside

computers, using our solution, the humans are forced to follow the computers, as the computers cannot automatically adjust to changes coming from the humans. Because of the different scenario, we don't know if the 30 milliseconds limit is still valid when humans play together with computers. It would be beneficiary to test our solution with actual musical artist performing alongside our solution to see if our assumption hold true.

7.1.3 Code Sharing Performance

A big surprise from our evaluation was that iOS devices have better synchronization characteristics under high traffic loads compared to low traffic; the iOS devices are almost eight milliseconds tighter under high traffic. This could indicate that the devices has aggressive power savings, especially in the network stack. One possible explanation would be that the device puts the network stack to sleep if it hasn't received a message in a few hundred milliseconds. In our low traffic scenario each device received a message at least every two seconds and somewhere in that interval the device starts power saving. In our high traffic scenario, each device received a message at least every 10 milliseconds and here the device did not seem to enter a power saving state. We don't know exactly why this is happening, but aggressive power-saving seems a reasonable explanation.

The implication is that even a tight ecosystem such as Apples have large variations in performance characteristics when shared code is run across different devices. Therefore, although we can achieve the same functionality from a shared code base we still need to do device specific optimizations.

7.2 Lifespan

Can our work stand the test of time? Will it be relevant in a few years or even beyond that? There is, currently, no other solution that tackles synchronization in a distributed manner, so it is relevant today. But maybe a company introduces a fully distributed synchronization, makers of DAWs such as Ableton, Logic and Pro Tools seems obvious prospects. Avid, the maker of Pro Tools does in fact have a somewhat distributed solution - but one that only supports a maximum of 12 devices. However, a limiting factor of their solution is that it is a closed solution; it only works with their own software not allowing the performers to choose which software they wish to use. We think that if the other DAW makers introduced a distributed synchronization system they would also feel tempted to create a closed solution, hoping to make their own software more appealing for end users.

An ideal scenario would be that a standardized solution was presented from a group such as the Audio Engineering Society (AES). If AES created a standard, it would be possible for every media software developer to adapt and implement their standard.

7.3 Problem Solved?

After spending six months thinking, implementing, rethinking, reimplementing our solution it makes sense to look at it again from a bird's eye view. We set out to create a synchronization solution for human performers to allow them to perform alongside computers on stage.

Our final solution requires a minimum of three nodes before the system starts to form a stable synchronization. This is a limiting factor because our experience shows that many systems only have two nodes in practice. Our expert panel have also mostly been involved in setups where two devices needs to be synchronized. Does this make our solution unusable or maybe just a bit over engineered? And could another approach have been taken?

We chose a tight synchronization model which requires at least three nodes as a basis for our algorithm, the reference broadcast synchronization [3]. However we could have chosen a less tight synchronization model requiring only two nodes such as NTP [8]. For the algorithm to work with only two devices using the tight synchronization would require that the network kernel of OS X and iOS would allow broadcast or multicast messages to be returned to the sender from the router and not by direct loop-back. The requirement for three devices unfortunately implies that we won't solve synchronization problems for all stage performers, only the ones who have at least three devices on stage.

7.4 Future Work

Our solution only target a subset of synchronization problems faced by musical artist, the problem of synchronizing playback of time-dependent media files. However, another interesting problem is how to synchronize real-time operations, such as playing externally triggered instruments like synthesizers. Using multiple instruments together creates synchronization problems, which leads to unwanted musical behavior.

Each instrument has its own delay from receiving a message until any output is generated. This delay would need addressing if we were to start solving this problem. The process of transferring the message from the sender to the instrument incurs another delay, this would also need addressing.

When instruments drift apart, even by small margins, the feeling of the music changes and usually in an unwanted fashion. Although there are solutions for this problem, like the SyncGen[1], they are usually either very expensive or difficult to setup and sometimes both.

Our solution cannot handle this scenario as we use a large buffer for commands, currently one second. Handling a real-time solution would require a much small buffer, preferably somewhere in the tens of milliseconds. Nevertheless, it would be an interesting extension if it were possible. A distributed synchronization tackling real-time and time-dependent scenarios, could help improve the work for musical artists in the creative studio phase as well as the stage-performing phase.

After our demonstration to our expert panel, we received valuable feedback. Expert one suggested building a modular synchronization infrastructure using our system, and have it generate different types of time code. This would be possible because we have already built the software in a modular fashion. The MTC generation is a module that is plugged into our synchronization algorithm, and it would be workable to add other types of generators such as Linear Time Code (LTC), video playback or anything really that is time dependent. LTC generation would facilitate the possibility of connecting our

[1] `http://www.innerclocksystems.com/`

system to external equipment like video servers, lighting desks, and many other types of professional hardware used for stage performances. Expert one uses LTC enabled equipment for several acts and would find this addition valuable. The same expert would also like to see a textual module using the same synchronization engine; this would make it possible to have text notes show up at specific points during a performance. An example could be that an iOS device was visible to the sound engineer mixing a show and notes like "add reverb to vocal" could show up at the correct time or it could be used for speakers who have to see their speech roll by as they speak.

Our algorithm should be altered on iOS devices to tackle the discovery that these devices works better under high traffic loads. Burst of traffic could be created before the synchronization pulses, because if the problem is in fact power a saving network stack this could be a possible solution that would wake up the network stack before it receives the important synchronization pulse.

Since we have based on algorithm on broadcast messages, we are bound to keeping nodes on the same subnet and this is an obstacle if you need to synchronize nodes across different networks. The algorithm could be altered to allow for multi-hop networks with nodes on different subnets, this would be possible by creating gateway nodes at each subnet who exchange information about their respective subnets. This would, in theory, make it possible to synchronize an unlimited number of devices, or synchronize nodes over larger distances.

7.5 Summary

In this chapter, we have talked about interesting findings from our evaluation. Maybe our measurement and comparison techniques should have been different, there is no evidence proving that we have chosen the correct approach. We have reflected upon our approach to our solution and found that we don't have an answer for every performer out there. Finally, we looked at future work for the project where our experts, also chimed in, with valuable feedback.

Chapter 8

Conclusion

Stage performers, such as musical artists, use a combination of human and computer actors on stage. All the actors need to execute in unison to create the envisioned performance. We have created a distributed software solution, which enables computer actors to perform in unison. The human actors can synchronize their actions against the audio signals generated by the computer actors completing the equation.

We set out to create a cross-platform solution, ideally sharing 100% code across different platform implementations. After preliminary trials of the Xamarin cross-platform environment, our ideal of code sharing was not viable. First, the Xamarin platform itself was not stable when the trials were performed. Second, the amount of shared code we could achieve was not satisfactory because the bulk of the code we needed required a different implementation per platform.

After our interviews with stage performance experts it became clear that the predominant platforms required are the ones provided by Apple. Therefore, we narrowed our platform choice to Apples OS X and iOS devices and then we could achieve our ideal of 100% code sharing across OS X and iOS using the programming language Swift.

The solution we have created brings research from the distributed systems community to the world of stage performances. We have managed to create a fully distributed synchronization setup that does not have a single point of failure, which is normally the case for synchronization on stage. Through a series of empirical evaluations, we considered different clock synchronization algorithms from the research community. However, none of them fulfilled our exact needs, so we created our own algorithm improving on the reference broadcast synchronization protocol (RBS) [3] by distributing the pulse broadcasts across all nodes instead of having a single node with that responsibility. In addition, we improved on the number of message exchanges required for synchronization; in a system with n nodes, RBS requires $O(n^2)$ messages whereas our protocol only requires $O(n)$ messages.

Our solution can be expanded with additional platforms, as long as it is a platform capable of UDP and TCP connectivity. Our system makes no internal assumptions about the platform and nodes communicate solely by message exchanges that could be sent by any platform.

In our evaluation, we tested our system on eight nodes, which was the number of devices we had available. The maximum number nodes it supports is limited by the bandwidth saturation of the network. The evaluation showed that our system could synchronize OS X nodes to be within four milliseconds of each other, iOS nodes are

slightly less precise with a precision of 11 milliseconds.

Gurevich et al. [4] has found that humans need to be less than 10 meters apart, the equivalent to a 30 milliseconds delay, for them to perform synchronized music together. Our evaluation shows that our system can go well below that threshold and should therefore be good enough for musical artists to perform with.

Our distributed system requires a minimum of three nodes before the system forms a stable synchronization. This is a limiting factor because our experience shows that many systems only have two nodes in practice and then our solution wouldn't fit the problem. To make our solution work on only two devices it would require either another synchronization algorithm, with a presumed lower precision, or that the network kernel of OS X and iOS would allow broadcast or multicast messages to be returned to the sender from the router and not by direct loop-back.

Our expert panel was very pleased with our system finding its abilities and performance characteristics valuable. One expert expressed the desire to get it installed as soon as possible for all his performers.

We conclude that we have solved the problem we set out to solve, which was to create a platform that allows humans and computers to jointly perform on stage. Moreover, we conclude that our solution is not final and that we will continue our work to improve it.

Chapter 9

Bibliography

[1] Committee. Resolution 1 of the 13th conference generale des poids et mesures (cgpm), 1967.

[2] Flaviu Cristian. Probabilistic clock synchronization. 3:146–158, 1989.

[3] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. 36:147, 2002.

[4] Michael Gurevich, Chris Chafe, Grace Leslie, and Sean Tyan. Simulation of networked ensemble performance with varying time delays: Characterization of ensemble accuracy. pages 1–4, 2001.

[5] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. 21(7):558–565, 1978.

[6] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal clock synchronization in networks. page 225, 2009.

[7] Miklós Maróti, B Kusy, Gyula Simon, and Á Lédeczi. The flooding time synchronization protocol. pages 39–49, 2004.

[8] D.L. Mills. Internet time synchronization: the network time protocol. 39(October):1482–1493, 1991.

[9] Su Ping. Delay measurement time synchronization for wireless sensor networks. 2003.

[10] Ill-Keun Rhee, Jaehan Lee, Jangsub Kim, Erchin Serpedin, and Yik-Chung Wu. Clock synchronization in wireless sensor networks: An overview. 9:56–85, 2009.

[11] Julien Ridoux, Darryl Veitch, and Timothy Broomhead. The case for feed-forward clock synchronization. 20:231–242, 2012.

[12] Philipp Sommer and Roger Wattenhofer. Gradient clock synchronization in wireless sensor networks. pages 37 – 48, 2009.

[13] By Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin. Timing-sync protocol for sensor networks. 47:34–40, 2006.

[14] Ichiro Ushijima, Masao Takamoto, Manoj Das, and Takuya Ohkubo. Cryogenic optical lattice clocks with a relative frequency difference of 1x10 -18. pages 1–20, 2014.

[15] Paulo Veríssimo, Luís Rodrigues, and Antonio Casimiro. Cesiumspray: a precise and accurate global time service for large-scale systems. 12(3):243–294, 1997.

[16] Geoffrey Werner-Allen, Geetika Tewari, Ankit Patel, Matt Welsh, and Radhika Nagpal. Firefly-inspired sensor network synchronicity with realistic radio effects. page 142, 2005.

Chapter 10

Glossary of Terms

- **API** — Application Programmable Interface.
- **DAW** — Digital Audio Workstation.
- **GCD** — Grand Central Dispatch.
- **GPS** — Global Positioning System.
- **LTC** — Linear Time Code.
- **MAC** — Media Access Control.
- **MIDI** — Musical Instrument Digital Interface.
- **MTC** — MIDI Time Code.
- **NIC** — Network Interface Card.
- **PCL** — Portable Class Library.
- **SMPTE** — Society of Motion Picture and Television Engineers Time Code.
- **TAI** — International Atomic Time.
- **UTC** — Coordinated International Time.

10.1 Time Abbrevations

- **ns** — Nanosecond - 1000000000 ns (1 billion) per 1 s.
- **μs** — Microsecond - 1000000 μs (1 million) per 1 s.
- **ms** — Millisecond - 1000 milliseconds (1 thousand) per 1 s.
- **s** — Second - 1 s per 9192631770 (9.2 billion) oscillations of the celsium atom [1].
- **min** — Minute - 60 s per 1 min.
- **hour** — Hour - 3600 s per 1 hour.

Appendix A

Produced Software

During the thesis process we created the following software.

1. **Simultanio** — Our proposed solution for synchronizing nodes.
2. **MTC Generator** — Generates MIDI Time Code.
3. **CocoaAsyncSocket**[1] — Objective-C Socket Library, modified to timestamp incoming packets as soon as possible.
4. **Time Code Tester** — Calculates the preciseness of incoming MTC.
5. **Network Measurements** — Calculates delay- and offsets between any two nodes and produces R plots from the calculation.
6. **ku-latency**[2] — Measures the difference in time between when the kernels pace gets a message compared to when the user space gets the same message. Modified to run on OS X.
7. **Xamarin Prototyping** — Sample project for testing the possibility of creating a crossplatform (Windows, OS X, Android, iOS) application from one codebase.
8. **Shell Scripts** — Variuos shell scripts have been created to avoid manual processing during the evalution process.

[1]Original version - `https://github.com/robbiehanson/CocoaAsyncSocket`
[2]Original version - `https://vilimpoc.org/research/ku-latency/`

Appendix B

Simultanio Software Architecture

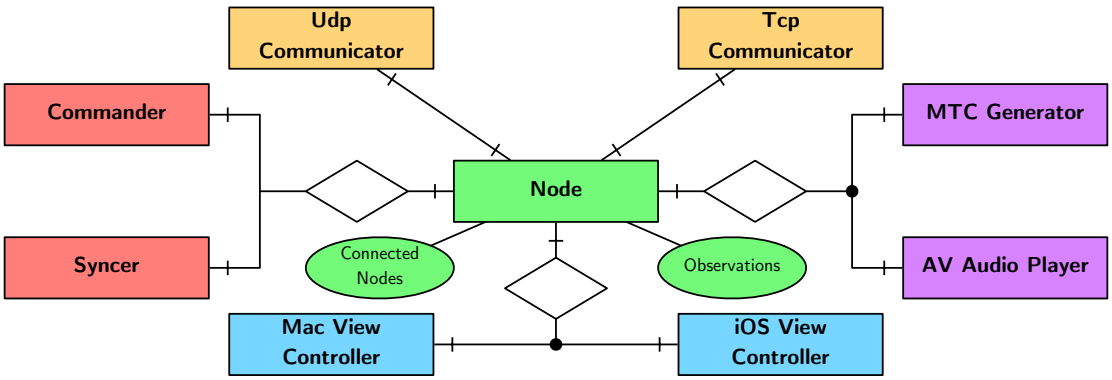


Figure B.1: Simultanio software architecture.