Princeton University
Department of Computer Science

Robust, Reliable, Real-time Networking for PLOrk (Princeton Laptop Orchestra)

Mark Cerqueira

Independent Work

Advisor: Perry Cook

4 May 2009

This paper represents my own work according to University Regulations.

Table of Contents

Introduction to PLOrk

The Princeton Laptop Orchestra (PLOrk) was co-founded in 2005 by Princeton professors Perry Cook and Dan Trueman. As a musical group, PLOrk aims to create music from the range of player's creativity, which has the potential to "both guide the development of new instruments and technologies" while also invigorating the concept of the orchestra [Trueman 2007]. Members of PLOrk use their own laptops and additional interface devices to play pieces that are coded predominantly in ChucK. During its flagship year, PLOrk was composed of fifteen members. Today, PLOrk has over thirty members and plays with well-known groups such as Matmos and So Percussion. Players work with 2.1 GHz Macbook laptops running Mac OS X and connect to an Airport Extreme Base Station that is used solely by PLOrk and does not connect to the Internet.

Introduction to ChucK

Most compositions performed by PLOrk are written in ChucK, a programming language designed for real-time music synthesis, composition, and performance which was developed by Ge Wang and Perry Cook in 2003. The language was designed to be flexible, allow control over the passage of time in programs, and allow synchronization among programs running at the same time, while maintaining a strong correspondence between code structure and timing. ChucK resembles Java, but the language relies very heavily on a unique overloaded operator called the ChucK operator: =>. Among its many uses, the ChucK operator is used to ChucK (i.e. throwing one entity onto another or assign) values to variables (e.g. 3 => int foo), and to connect unit generators in a sequential manner (e.g. SinOsc s => Gain g => dac).

While many programming languages allow the programmer to specify *what* a program will do, ChucK also allows the programmer to specify *when* a program will do certain things, making ChucK a strongly-timed language. ChucK has two primitive types, time and duration, that allow fine control over time. Time itself is a primitive type and computable – the keyword *now* specifies the time (in samples) since a program began running. A duration is a period of time that is specified by value::units (e.g. 1::second is a duration of 1 second). Computations are assumed to happen infinitely fast with respect to the program running, so time only advances in the program when it is explicitly advanced, which is done by ChucKing a duration to the keyword *now* (duration => now). By advancing time, the program pauses for the amount of time specified in duration and control is given to the ChucK virtual machine and synthesis engine, which produces sound. Time can also be advanced by waiting on an event, which can be a signal sent out by another ChucK program running on the same machine, data from a human interface device, or a message sent over the network from another program [Wang 2008].

Networking for PLOrk – Problem/Motivation

Like the conductor of an orchestra, the network is a powerful tool that can be used to create a particular type of sound from the orchestra as a whole. But unlike a traditional conductor that typically keeps players on beat, controls dynamics, and cues sections in, the network also allows any type of information to be transmitted to all players, such as filter parameters, text messages for players to read, or control messages to manage the volume levels of players. Depending on the design of a particular composition, the network may not be used at all and a person conducts the orchestra in the traditional sense; a program can be set to conduct the orchestra automatically over the network; or a person can control a program that automatically keeps tempo but allows the conductor to adjust other settings [Cook, et al. 2008]. Most of the

current PLOrk repertoire relies on networking for either tempo synchronization or transmission of other performance-related information.

During its earlier years, when there were fewer members, the ability to synchronize PLOrk orchestra over a wireless network was described by early composers as "remarkable, though not flawless" [Cook, et al. 2008]. With a single computer acting as a conductor sending out pulses for players to synchronize to, pulses could be sent out as frequently as every 40 milliseconds without a problem. Still, there were times when the network presented difficulties, and those involved in PLOrk knew exploring more robust and reliable means for communicating was important for compositions to function as designed [Cook, et al. 2008].

With the growing membership of PLOrk, more and more hiccups in the network are occurring and compositions that used to function properly over the network, now do not perform as well. With more traffic being transferred over the wireless local area network (WLAN), the increased traffic has caused packets to experience higher delay or be dropped completely. In pieces where a server sends a pulse over the network for all players to synchronize to, the aforementioned issues result in computers synchronizing to different pulses at different times, which produces an audible and undesirable variation in the beat players are synchronized to.

There are workarounds to using flaky wireless networks; such as having players manually sync their computers during a piece and then performing the piece as usual. Still, having a reliable network to automate synchronization allows players to focus more on performing and less on making sure their computer is synchronized. Manual synchronization requires a person to play the traditional role of an orchestral conductor, which is not atypical for PLOrk, but should not be a requirement of all pieces. Wired networks using Ethernet and hubs to connect all of PLOrk may also be an option to explore, although the directors of PLOrk strongly prefer the

current wireless setup. A more desirable solution to dealing with the inherent unreliability of networks is to design a protocol that makes best-effort delivery more reliable and robust.

Current Networking Protocol – Open Sound Control (OSC)

For communication over networks, ChucK currently implements Open Sound Control (OSC) - a protocol designed for communication among devices over networks to produce interactive music. There are many implementations of OSC in all types of programming languages, interactive sound synthesizers, and sensor/gesture capture hardware. All of them enable devices to communicate over existing network technologies – communication is most important while reliability, accurate timing, and accepting various data formats to transmit are desired features [Wright 2005]. The ChucK implementation of OSC contains two classes: OscSend for constructing and sending OSC packets, and OscRecv, which receives OSC packets and parses the contained data. The types of data that can be transmitted are integers, floats, and strings. OSC packets are transported over the User Datagram Protocol (UDP) [Wang 2006].

UDP is commonly used for time-sensitive applications such as VoIP and online gaming. In UDP, packets are simply sent to their destination – there is no acknowledgement from the receiver that the packet has been received and packets are not resent if they are dropped. For time-sensitive applications, it is not worth sending a packet again if it does not reach its recipient because if the packet was resent, by the time it reaches the recipient, the information contained in the packet would be old and obsolete. Consider a call over voice over IP (VoIP) – packets containing voice data are sent from caller to caller. If a packet is delayed or dropped it is not worth sending the packet again since that voice data is old.

Although networking for ChucK is a time-sensitive application, UDP is not the perfect solution for our problem especially considering how small the PLOrk network is. The other main

protocol for transporting packets is Transmission Control Protocol (TCP). TCP is designed for guaranteed delivery rather than the timely but inconsistent delivery UDP provides. Still, TCP has features that do not make it a good candidate for transporting our OSC packets. TCP resends packets that are dropped or lost which is not useful for real-time music production. TCP also requires a handshake between communicating parties, which is not entirely useful either – we can already create a pseudo-handshake between server and clients with OSC. Still, TCP has some features that would be useful to have running on the PLOrk network – mainly a higher guarantee of delivery of packets, discarding of duplicate packets, and flow control to avoid congesting the network. Developing a protocol that is as robust, and functions in real-time as UDP, but provides some level of guarantee like TCP is desired for the problems occurring in many PLOrk pieces and in music performance over networks in general.

Background Work

The concept of performing music over networks has become a widely researched area as the increasing power of networks has removed the traditional barriers performers had to share when performing together. But is performance over networks that are only best-effort and do not provide any quality of service guarantees even possible?

The SoundWire group, a research group based at Stanford University, has been researching how to transfer audio and perform interactive music over networks. In 2000, the SoundWire group put on several teleconcerts by streaming high-quality uncompressed audio between two musical events at separate locations on Stanford's campus. Later in the same year, the group was successful in putting on another teleconcert between Stanford and North Carolina over Internet2 [Chafe, et al. 2000]. The SoundWire group also turned the network between locations into an instrument by measuring network statistics between those two locations and

then transforming that data into sound, which provided a quick way to assess the quality of the connection [Chafe, et al. 2001]. Tests were also conducted by the SoundWire group to assess the effect of delay on two performers' rhythmic accuracy when they were in separate rooms. When separated by longer delays, tempo deceleration resulted, while shorter delays (< 11.5 ms) produced tempo acceleration. A delay of about 20 milliseconds between locations was found to be optimal for ensemble performance [Chafe, et al. 2004]. The early work done by the SoundWire group illustrated that performance over networks was certainly possible.

In the Gigapop Ritual performed in 2005, researchers from Princeton University in New Jersey, USA and McGill University in Montreal, Canada set out to perform jointly while being at the two distant locations to test if interactive performance over networks was possible. Although the Gigapop Ritual and the resulting framework that was designed, GIGAPOPR, could transmit audio, video, and MIDI data, the design principles behind GIGAPOPR attempted to work around many of the networking issues that occur when PLOrk laptops are communicating data over a local wireless network in the same room.

GIGAPOPR was designed to be straightforward while providing optimizations for operating over the low-latency, high bandwidth Internet2 and CA2Net networks. All data transmitted in GIGAPOPR used the User Datagram Protocol (UDP). While UDP does not provide flow control or congestion control, the designers of GIGAPOPR realized that waiting for the retransmission of dropped or delayed packets that TCP implements would not be useful in a live, real-time performance. Packets sent via the GIGAPOR framework also included a sequence number in the header that enforced ordering of incoming packets, allowed detection of packet loss, and made redundant transmission of packets possible. With sequence numbers it was

possible to send copies of each packet to increase the chance at least one of the packets would reach its destination [Cook, et al. 2005].

With any framework, there always exists some latency between sending a packet and the receiver of a packet because there is a limit on how fast packets can move – the speed of light. During this particular performance between Princeton and Montreal the average measured round-trip latency was between 120 and 160 milliseconds. Since there was no way to have a true real-time feed between the two locations, when performing a piece, one location served as the leading side and the other location served as the following side separated by the round-trip latency of 120 ms. In this particular performance, performers in Princeton served as the leaders and once the data arrived at McGill, the performers played along to what they were seeing. The researchers who developed the GIGAPOR framework concluded that performances relying on a best-effort network was certainly possible and was a concept worth pursuing especially considering how fast networks have become [Cook, et al. 2005].

In April of 2008, the SoundWire group set out to put on a joint real-time performance of Terry Riley's *In C* between Stanford University in California and Peking University in Beijing, China. With such a vast distance separating the two locations, many of the same issues present in the Gigapop Ritual had to be dealt with for this performance, which was designed to transmit both audio and video feeds. Like in the Gigapop Ritual, researchers dynamically measured the RTT between Stanford and Peking University and then used that measurement to set the tempo of the eighth notes that comprise *In C*. When a note was played at Stanford, it was transmitted over the network and then played at Peking exactly one eighth note later (relative to the performance at Stanford) and vice versa for notes played at Beijing. This technique of using the RTT between the two locations to set a tight rhythmic alignment between vastly separated

locations is known as feedback locking. The end result is a tightly synchronized piece at each of the performance locations, which allowed there to be a live, interactive transcontinental performance [Cáceres, et al. 2008].

While the GIGAPOPR framework and SoundWire performance of *In C* were synchronizing over rather large distances, some of the problems the researchers worked around still need to be addressed for performers who share the same room on a local network. These performances and frameworks were also designed to transfer audio and video feeds which is not currently used in any PLOrk pieces. Researchers at Waseda University in Japan designed a protocol that applies more closely to the type of protocol that may be used in PLOrk. In a typical PLOrk piece, control messages and synchronization pulses are the most common type of traffic transmitted over the network. The researchers at Waseda developed a protocol, called the Remote Music Control Protocol (RMCP) that integrated MIDI and computer networks.

RMCP is a connectionless server-client model where various clients broadcast messages that various servers receive and process. By relying on broadcasting, information is shared among all servers without having to specifically retransmit packets to each server. Each server serves a different, specified role – one may be visually displaying what is happening, and another may be producing the sound [Goto, et al. 1997]. The most interesting feature of RMCP lies in the time scheduling feature built into it.

RMCP packets have the option of including a timestamp in the packet header. If a server receives a packet before its timestamp, the packets are queued and only processed when the current time matches the timestamp. Packets received after the time in the timestamp are discarded and packets without timestamps are processed immediately. Timestamps were included to compensate for variation in network latency. This implementation requires that the

clocks among all computers be synchronized. In RMCP, synchronization of clocks occurs via the RMCP Time Synchronization Server (RMCPtss). Essentially, all machines keep a table of the offset of their clocks to every other machine. Periodically, a machine broadcasts what time it is and all other machines then calculate their offset to that machine and update their table of offsets. With RMCPtss, time among all machines becomes absolute and timestamps can be used [Goto, et al. 1997]. The scheduling functionality present in RMCP may potentially be a useful model to emulate in a PLOrk setting to deal with jitter in network latency.

Roger Dannenberg of Carnegie Mellon University developed another system for real-time distribution of data over networks called Aura. Aura was designed to be flexible in the data it can transmit while providing low-latency, real-time communication between clients. The most interesting aspect of Aura is that, unlike previously described frameworks, Aura runs over TCP instead of UDP. While UDP seems to be the best protocol for real-time transport, it does not guarantee delivery of messages. Dannenberg found UDP to be reliable across local area networks in controlled situations, but found that multiple machines transmitting messages via UDP resulted in dropped packets. Because of the reliability issues with UDP, Dannenberg opted to try using the reliable protocol for packet transport, TCP [Dannenberg 2001].

TCP is not the preferred protocol for real-time systems: TCP buffers information to minimize the number of packets that need to be sent which creates a delay that is separate from the delay packets experience over the network; TCP also retransmit lost or heavily delayed packets which is not useful since the data that packet contains will most likely be obselete. Still, with a few option changes to TCP, Dannenberg was able to make TCP's timing behavior very similar to UDP's. By turning off the option to wait for more data to come in to merge so fewer packets are sent, more packets are sent, but there is no delay between sending data and having it

sent. To make retransmission of lost packets not obsolete in his real-time system, Dannenberg sent more audio samples than what was typically computed in the period between when packets are sent. If a packet dropped, there would still be data to process while the next packet was sent [Dannenberg 2001]. The success of Aura as a real-time music processing system and its use of TDP helps show that one need not create a real-time system at the cost of reduced reliability – you can have both real-time and reliable communication over a network.

Packet Loss and Music Performance Over Networks

While UDP appears to be the preferred protocol to use when implementing tools to perform music over networks, past implementations added extra functionality on top of UDP to make it more reliable. The features added to UDP did not make it any less capable to perform real-time communication over a network, but added mechanisms to ensure some level of reliability. In the new protocols drafted for networking in PLOrk, the general design goal was to add functionality to the UDP-based OSC classes already present in order to make them more reliable and robust, while ensuring real-time performance was still possible.

The current networking protocol used in ChucK has no mechanism to detect packet loss. Simply adding sequence numbers to every packet sent, like in the GIGAPOPR framework, allows a receiver to determine many things about arriving packets. If the receiver keeps track of the sequence number of the last packet received from every client, any packets received with a sequence number less than the last observed is old and should be discarded. Packets received with a sequence number greater than the last sequence number plus one means a packet was lost. Including sequence numbers also allows redundancy to be implemented since a receiver can sort out redundant packets by simply keeping track of the last sequence number seen and discarding

redundant packets that share that same sequence number. The GIGAPOPR framework designers mentioned the possibility of using redundancy to increase the chance of packet delivery.

If sequence numbers are used and redundancy is implemented, implementing more functionality to control how redundant the protocol is should also be considered. If there is not enough redundancy and packets are lost, a mechanism should be in place to increase redundancy. If there is sufficient redundancy, the protocol should be able to back off and behave like a good neighbor to avoid congesting the network with unnecessary redundancy. With these principles in mind, I drafted a new protocol for networking in ChucK called Adaptive and Redundant OSC (AROSC).

Adaptive and Redundant OSC (AROSC)

AROSC is an extension that is built on top of the two OSC classes in ChucK, OscSend and OscRecv. These extensions were built and coded in two new ChucK files: AROscSend, AROscRecv. All methods available in the OscSend/OscRecv have their counterparts in the AROSC version so adapting the code is relatively easy (see the converting code from OSC to AROSC guide in the Appendix for more information regarding code converting). AROSC adds three data fields to every packet – the source host (string), a sequence number (integer), and the sending rate (integer) as shown in Figure 1.

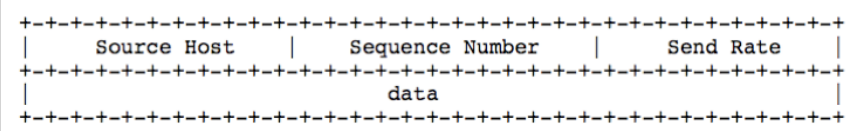

Figure 1 – AROSC packet

Since the PLOrk network is relatively small, AROSC's first feature is sending multiple copies of a packet via AROscSend objects in hopes that sending multiple copies of a packet over the network will improve delivery rate. If one packet is sent over OSC, there is a risk that packet

will be dropped or delayed. In AROscSend, multiple copies of the same packet are sent to the destination with the hope that at least one copy will make it to the destination. AROscSend currently defaults to sending ten copies of each packet and has a method that allows the user to change the default redundancy rate. Since the network is relatively small, this added redundancy will hopefully provide a higher level of guarantee for packet delivery, rather than exacerbating packet loss because of increased traffic.

For every packet that AROscSend sends, each unique packet is given a sequence number, which increases sequentially. Every packet also includes the name of client sending the packet. AROscRecv keeps a list of all clients and the last sequence number received from those clients in an associative array. When a packet is received, the sequence number is used to check if the packet is new or a copy of one already received. Duplicate packets are discarded (a feature of TCP). Packets that are old (sequence number is less than the last sequence number seen) are also discarded. With sequence numbers, data that arrives out of order is better handled (another feature of TCP).

Sending multiple copies of a packet may not be enough – what if all the packets are dropped? The regular OSC class has no mechanism for checking for dropped packets. AROSC uses sequence numbers to check for dropped packets. If a packet is received from a client and the sequence number is more than one greater then the last seen sequence number, a packet must have been dropped so a message requesting a rate increase is sent to the client. This message is sent multiple times to ensure delivery and has a large random integer attached to it so the client can discard duplicate requests.

The last piece of data attached to every AROSC packet is the copy rate – how many copies of the packet are being sent. AROSC can adapt to send more packets, but it should also be

able to cut back when most packets are getting through to aid in flow control. This good citizen policy helps reduce packets being delayed or dropped by reducing unnecessary redundant traffic. To implement this, AROscRecv keeps track of how many times it has seen a duplicate packet – if that number matches the number of copies sent, a message is sent to the client to cut back on the sending rate. Currently, the minimum sending rate allowed in AROSC is three copies; any requests to lower the rate when AROscSend is only sending three copies of each packet will be ignored. Figure 2 summarizes how AROscRecv handles arriving packets.

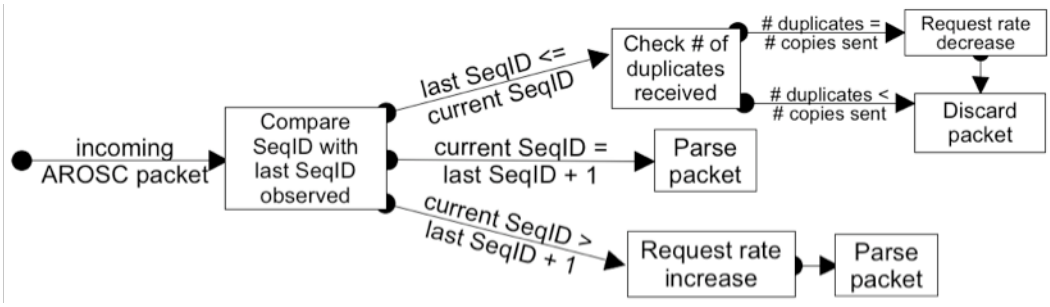

Figure 2 – AROscRecv handling incoming packets

While still being UDP-based, AROSC implements some TCP-like features that provide a higher guarantee of packet delivery. While there is a possibility that the increased traffic brought about by redundancy will only make networking worsen, the adaptive measures built into AROSC attempt to prevent this from happening. And while the redundancy implemented in AROSC helps with dropped packets, the variable delay packets experience on the network may still be an issue that must be addressed.

AROSC Testing

AROSC was tested and compared to OSC during a PLOrk rehearsal. For testing purposes, two programs were written to test AROSC and OSC under various conditions. A server program was written that ran on a single machine. This server periodically broadcasted

messages via multicast with messages that commanded the client programs to send traffic over the network to the server. The messages sent from the server included which protocol to use (AROSC or OSC), how many packets to send, and how often to send them. When the clients received this message, the message would be parsed, and the specified number of packets at the specified rate were sent to the server. Each client's machine name and Sequence IDs were included in each packet so delivery success could be assessed for each client. The server receiving the packet logged the name of the machine sending the packet, the sequence ID, and the time the packet was received. Various sending rates were tested – sending a packet every 500, 250, 100, 50, 25, and 10 milliseconds.

Packets sent at rates ranging from a packet every 500 milliseconds down to a packet every 100 milliseconds performed equally well with both the OSC and AROSC protocols. With both protocols, packets arrived in order at the server and none were dropped. As the rate packets were being sent increased there was a noticeable increase in the variance in delay observed at the server receiving packets from all the clients. This could be attributed to the increase in traffic over network that increases the delay some packets will experience as they travel over the network.

Packets sent at and below rates of a packet every 50 milliseconds via AROSC did not perform better than OSC. Because of the increased load on the server receiving redundant AROSC packets, the server had to do more work for arriving AROSC packets compared to OSC packets. Due to the increase in traffic caused by the redundant nature of AROSC, the AROSC receiver had to perform a larger amount of calculations on arriving AROSC packets in a shorter amount of time. At higher sending rates, the redundancy feature built into AROSC bogged down the server program and made networking among machines unusable.

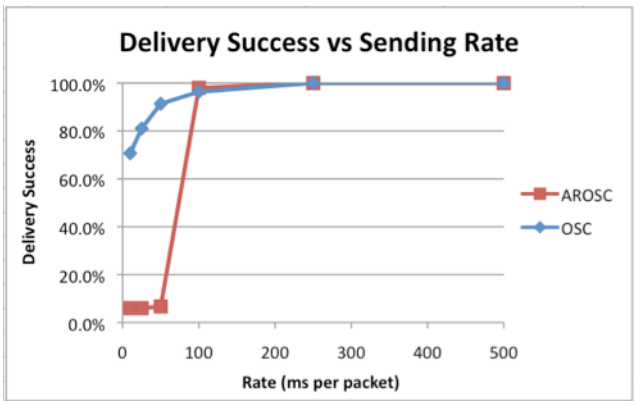

Figure 3 – AROSC/OSC testing summary

Figure 3 summarizes the AROSC testing results. At each of the rates tested, clients sent thirty packets to the server at the specified rate. Delivery success was defined as a packet arriving in order relative to the last received and in a timely manner – packets were given three times the rate they were being sent at to arrive after the last packet was received, before the packet was counted as not having arrived successfully. For AROSC packets sent at rates faster than a packet every 100 milliseconds, only a small fraction of packets that were the first to be sent were delivered successfully before the server became overloaded by the redundant packets and unable to process all incoming packets in a timely manner. At the same rates, OSC packets were delivered succesfully at much higher rates, suffering only from occasional reording of packets while being transmitted and congestion that caused packets to arrive too late.

Although AROSC did not perform well under high sending rate conditions it did show that redundancy is not a necessity for the network setup it was tested on. Packets sent over OSC or AROSC were rarely completely dropped – in the worst case, a packet just experienced an incredibly high amount of delay and arrived out of order at the server. If dropped packets was not the main issue affecting PLOrk compositions, other properties of networks were analyzed and adapted around to get better networking for PLOrk.

Packet Delay and Synchronization

Although AROSC intended to solve the problem of dropped packets over a network, that problem was not the main issue affecting PLOrk pieces. Another inherent property of networks – variable latency in packet delivery – presents greater issues for PLOrk pieces that rely on networking. The delay packets experience would not be an issue if it were always constant – when a server sends out pulses for synchronization, if all those pulses took the same amount of time to reach all clients, every client would be in sync. The jitter present in delay is what causes an issue – if a server sends out pulses to many clients and each pulse takes a different amount of time to reach each client, clients will be synching to the same pulse at different times. A protocol that works around the inherent delay and jitter of delay present in networks is desirable in these circumstances.

The current implementations of the receiving classes of OSC process packets as soon as they arrive in the buffer. When a packet is sent from one client to another, the receiver processes it as soon as the packet arrives after travelling over the network. Since ChucK is strongly-timed, it would be rather simple to send a timestamp field and have a receiver parse a packet only when the current time is the timestamp. The problem with this implementation is that there is no notion of an absolute clock among different machines or even different shreds (an instance of a ChucK program) running on the same machine. ChucK bases its time (the value of *now*) on the number of samples that have passed since the shred began running so the notion of time depends only on when the shred began running. If a timestamp field were to be properly implemented, there also needs to be measures to synchronize the time among all machines so the timestamp is treated the same across all machines. A measure of latency between machines also needs to be measured so

synchronization of clocks can account for the varying latency in packets communicating the synchronization data.

Synchronizing system clocks over a network has already been studied and implemented in the Network Time Protocol (NTP). In NTP, a system clock is synchronized to a reference clock that provides the current time. The protocol consists of multiple exchanges of request and replies from client to server to synchronize the system time of the client to that of the server. The client first sends a request to the server with its own time (originate timestamp) in a packet. When the server receives this packet, it attaches its own time (receive timestamp) and then attaches the time when the packet is sent back (transmit timestamp). When the client receives this packet, the client logs when the packet was received so it can calculate the delay in receiving the packet to provide more accurate synchronization. It normally takes about five exchanges of this type to fully synchronize. When synchronization that takes place over the Internet, the typical accuracy comes within 5 to 100 milliseconds of the actual time on the reference clock [Windl, et al. 2006].

Adrian Freed, a researcher at the Center for New Music and Audio Technologies at UC Berkeley, has been researching how to improve OSC to deal with the issues jitter in latency presents. Freed notes that although there are many implementations of OSC, most lack a mechanism to manage the delays and randomness present in networks [Freed, et al. 2008]. Freed proposes that, assuming the communicating machines' system clocks are in sync (via NTP or some other synchronizing protocol), a sender can add a time-tag to an OSC packet, which the receiver queues upon arrival and only processes it at the time marked for evaluation that is attached to the packet. Freed calls this setup forward synchronization, as communicating machines are synchronizing to a future point rather than synchronizing immediately upon packet

arrival. Freed notes that how far in the future the time-tag should be set depends on network latency which may vary from network to network, but if an NTP-like construct is used to synchronize the clocks, this issue becomes less of a concern as NTP keeps a measurement of average latency [Freed 2004].

An OSC implementation that could synchronize clocks of communicating devices and implement a time-tagging system could effectively curtail the issues presented by variable packet latency in networks. If the protocol used to synchronize clocks is strong enough and the time-tags used are greater than the largest observed latency, then pulses sent out from a server should be executed by all the clients regardless of the delay the packets experience reaching their destination. Based on these design goals, I drafted a new protocol for networking in ChucK called Time-Tagged OSC (TOSC).

Time-Tagged OSC (TOSC)

TOSC is an extension of the current implementation in OSC designed to work around the problem of variable latency packets experience over a network. TOSC compromises two modules – the first is an NTP-like protocol that synchronizes the ChucK virtual machine time among the communicating machines and the second is a forward synchronization model that is used to synchronize packet delivery to a future point so that all machines process packets at the same time. The TOSC protocol comprises four classes – two that implement the NTP protocol, NTPServer and NTPClient, and the other two that mirror the OSC classes already present in ChucK – TOscRecv for receiving packets and TOscSend for sending packets. When using the TOSC protocol, one machine functions as the NTP server and runs NTPServer while all other machines run NTPClient. TOscRecv and TOscSend then use the synchronized clocks provided by the NTP classes to schedule packets for future processing.

In the NTP protocol implemented, NTPClient objects communicate with a single NTPServer object and sync their clocks to the server's time. The typical synchronization transaction is summarized in Figure 4. The client program polls the server every second so it can stay accurately synchronized. The synchronization protocol begins by the client sending its system time (T_{client}) in a packet to the server. When the server gets this packet, it appends its system time (T_{server}) and sends the packet back to the client. When the client receives this packet it logs the time it receives the reply ($T_{received}$).

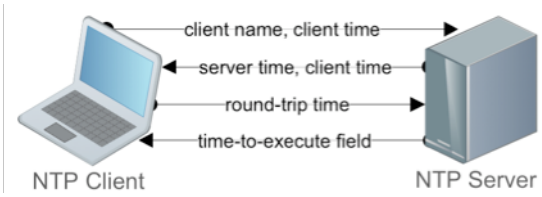

Figure 4 – NTP Synchronization Transaction

Simply adjusting the client's system clock to account for the difference between the two machines ($T_{server} - T_{client}$) does not account for the latency in the transmission of this information over the network, which changes over time depending on traffic. By logging the time the request is sent and then received, a pseudo-ping is implemented and the round-trip time for the packet can be calculated ($T_{received} - T_{client}$). Assuming that the total delay is split evenly between sending the packet to the server and getting the reply from the server, the delay from the server to the client and vice versa is half the round-trip time. Based on this information we can calculate the offset needed to synchronize the client's clock to the server's clock: ($T_{server} - T_{client}$) - (1/2) * ($T_{received} - T_{client}$). The current TOSC implementation I have written takes the most recent polled synchronization information and uses that to synchronize the clock of the client. Future implementations may use an algorithm, like NTP, to calculate an average so that more accurate synchronization can occur after enough samples are gathered.

The implementation of NTP also sends the time the server receives the packet in addition to the time the server sends the packet back to the client, which then allows the client to account

for the time the server was handling the packet. This was found to be unnecessary in TOSC because the code executed between the server receiving a packet from the client and sending it back occurs infinitely fast, so no time elapses in the ChucK virtual machine while the server processes a request. Thus, there is no need to include this time when synchronizing the clocks.

Once the system clocks are synchronized among communicating machines, packets sent can use the synchronized time provided by the NTP objects. In addition to appending the current time to every packet, the sender also needs to specify the time-to-open (TTO) offset. When the receiver gets the packet, it will calculate the time-to-execute (by adding the current time and TTO fields) and then queue the packet and pause the program for the duration between the current time and the time-to-execute before processing the packet. The main issue now is how large to make the TTO field and how we should go about determining it so that the protocol functions. If the TTO field is too small, the delay the packet experiences over the network may cause the packet to arrive too late. If the TTO field is too large, the rate of information flow will be less than the network can handle.

Fortunately, in the implemented NTP protocol, NTPClient objects measure the round-trip time (RTT) from client to server. If the latency is split evenly from client to server and from server to client, then the smallest acceptable value for the time-to-execute is half the RTT. Still, the latency may vary between the NTP server and the various clients synchronizing to it. While it may seem natural to have the NTPServer object running on the computer that is sending the packets, any machine should be able to send TOSC packets. In addition to an absolute notion of the current time, there also needs to be an absolute notion of what the TTO field should be so that regardless of who is sending packets, the TTO field will be the same.

To implement this, NTPClient objects report the RTT they calculate when synchronizing their clocks, to the NTPServer object. The NTPServer object takes all this information and uses it to calculate an appropriate value for the TTO. A large, safe value for the TTO is hard-coded into all NTP objects, and this value is used until the NTPServer object determines it is safe to change the value. Since all computers on the PLOrk network are one hop away from the server and each other, a single large value that is greater than the average observed RTT is used. The mechanism to change the TTO field across all machines is conservative when the field is being lowered – if the average reported RTT is 20% of the current TTO being used, the TTO value is cut in half. The mechanism is liberal when the field is being raised – if the average being reported rises to 60% of the current TTO, it doubles the value. Although cautious, this mechanism will ensure that any lowering of the TTO field will not be immediately followed by a need to raise it. With this mechanism, all machines communicating will have a safe and synchronized value for the TTO field.

With the clocks and acceptable TTO fields synchronized among all machines by the NTP protocol, the functions of TOscSend and TOscRecv are simplified. TOscSend appends the current time and the TTO offset in every packet. It would be possible to just add a single field and put the absolute time to the process the packet in it. Separating the current time and TTO offset fields was a design decision to make testing easier as both fields are computed separately in the NTP implementation. When TOscRecv gets a TOSC packet, it calculates the time the packet should be executed by adding the current time and TTO fields. If the time calculated is in the past, the packet is discarded. If the time is in the future, TOscRecv halts the program and advances time until the packet is scheduled to be opened and then passes along a signal for

packet parsing to begin. Figure 5 summarizes how TOscRecv handles incoming TOscSend packets.

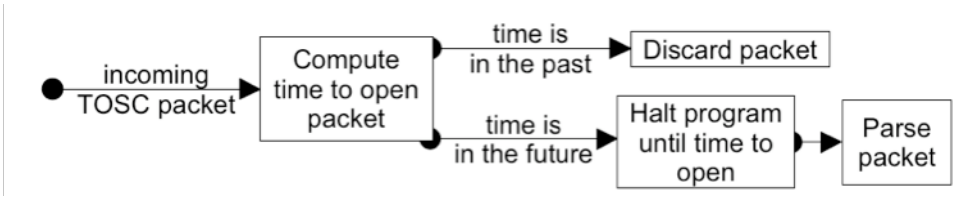

Figure 5 – TOscRecv handling incoming packets

With this TOSC implementation the issues presented by variable network latency become less of an issue. Since packets are scheduled for opening rather than being opened immediately upon arrival, if clocks and average latency can be measured and synchronized accurately, the scheduling mechanism will be strong enough that synchronization can occur among all the machines performing over the network.

TOSC Testing

TOSC was tested and compared to the OSC implementation during a PLOrk rehearsal to see if it could better handle the issues presented by varying latency. The test consisted of a computer functioning as a server that sent out packets via multicast to all other computers connected to the PLOrk wireless network. Upon receiving these packets, the clients would play a note on a mandolin unit generator. The test was run with both the original OSC implementation built into ChucK and the new TOSC protocol. Packets were sent with each of these protocols at varying rates: a packet every 500 milliseconds, 250 milliseconds, and 100 milliseconds. The server shred was killed completely, and there was a pause between running tests to ensure no packets from one test leaked into another. These tests were recorded for later evaluation and presentation. These recordings can be accessed at www.princeton.edu/~mcerquei/networking.

Controls were also added for comparison – these were produced by recording the output of a simple ChucK program that played a mandolin unit generator at the specified rates.

Evaluation was done by comparing the recorded test to the control of the same rate in the wave editor Audacity. Recordings were edited so that the first recorded pulse lined up the first pulse of the controls. An evaluation was done by playing both audio tracks simultaneously and noting if there was any deviation between the recorded test and the control. For more testing, the gain of each track was manipulated during the evaluation to bring out or subdue the sound of one track – this ensured that both tracks were being heard. Each track was also assigned to particular headphone speaker (i.e. the recorded test was played out of the headphone's right speaker and the control was played out of the headphone's left speaker). By separating the recordings being compared, it became significantly easier to evaluate the recorded tests.

At all three rates, the TOSC implementation produced better results when it came to synchronizing the notes clients were playing. Even at the slowest rate tested, a packet sent every 500 milliseconds, the varying delay over the network and OSC's inability to handle this variance produced a syncopated sound. Compared to the control, the syncopation among groups of pulses became even more apparent. At 250 milliseconds, a consistent beat is produced with OSC among all the clients that played for 12 pulses. The following 13^{th} pulse played sooner than 250 milliseconds later and this pattern repeated for the entire test. Further investigation into this consistent irregularity is worth future exploration. At 100 milliseconds, the OSC implementation played at the proper rate for about 12 seconds until the network became congested and the rate slowed down considerably for 3 seconds. After the congestion cleared the rate returned to normal. The TOSC implementation did not produce any large noticeable variance among clients

playing notes. When compared to the controls, the TOSC recordings matched up closely and played along without any serious audible variation.

An example of these findings is shown in Figure 6. This figure is a screenshot of the Audacity with three tracks in it – from top to bottom, the control recording, the TOSC recording, and the OSC recording all at the rate of a packet every .50 seconds. These tracks are being displayed in the FFT spectrum view mode, which shows the position of the energy of frequencies over time. This view mode was preferred over the waveform view because of the ambient noise in the recordings. Notes being struck on the mandolin are represented by vertical lines in the spectrum. Comparing the control track (topmost) to the TOSC and OSC recordings (middle and bottommost respectively) one can see that the plucks on the mandolin via TOSC are very synchronized to the control while the OSC recording fails to produce consistent plucks at the correct rate.

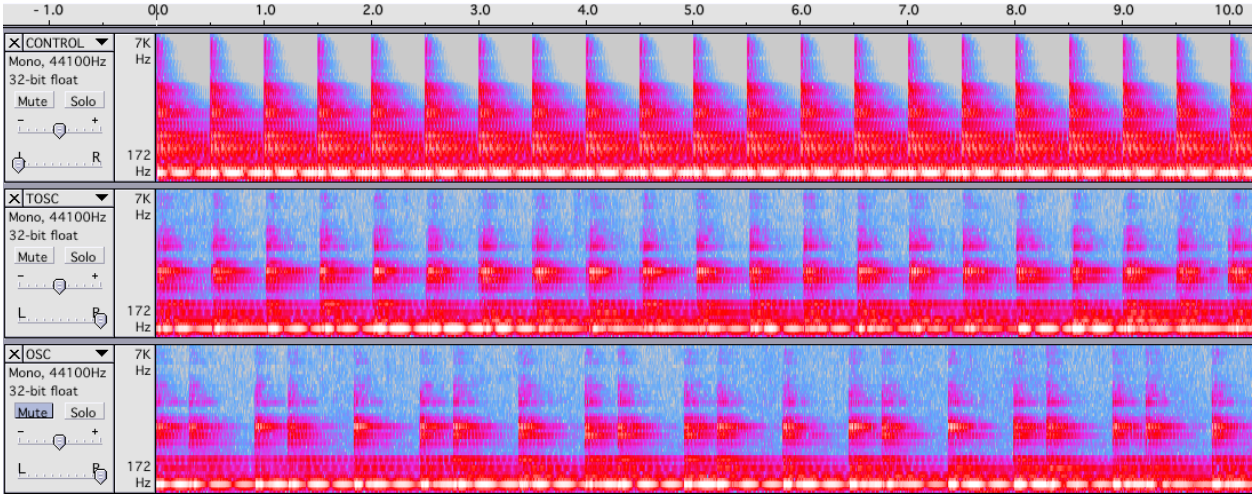

Figure 6 – FFT spectrum view of (top to bottom) the control, TOSC, and OSC recordings

The one notable issue that is not possible to hear in the TOSC test recordings is that some machines would intermittently stop playing for a while before returning to playing again. This issue came as no surprise because TOSC was designed to discard packets that arrived past the

time to parse the packet. The cause of this issue is the NTP protocol implemented is not finding an acceptable value for the TTO field in every packet. Implementing a more conservative algorithm that calculates a larger TTO field from the round-trip time information it receives will help ensure that fewer packets arrive to clients with expired timestamps. Even with this issue, the clients that did play were always synchronized and the clients that stopped playing were able to play in sync upon returning to playing.

Conclusion and Future Work

The work performed to bring more reliable, robust, and real-time networking to the Princeton Laptop Orchestra (PLOrk) brought about two protocols that addressed inherent issues of best-effort networks. Adaptive and Redundant OSC (AROSC) attempted to address the unreliability of networks and the issues caused by dropped packets. Time-Tagged OSC (TOSC) attempted to address the issues caused by the variance in delay packets experience while travelling over a network. Although the AROSC implementation did not prove to be a better alternative to the OSC implementation during testing, it eliminated the number of issues that needed to be addressed by TOSC. In real-world testing, TOSC proved to be better in synchronizing PLOrk by handling the issues caused by variance in packet delay.

Although TOSC performed better than OSC in a PLOrk-based setting, there is still much work that can be done to improve TOSC to make it even more robust and reliable while still providing real-time functionality. The NTP-based protocol that synchronizes clocks in the TOSC can be improved by implementing better algorithms to calculate achieve even higher accuracy for synchronizing clocks and getting better real-time measures of the latency packets are experiencing. The current NTP implementation in TOSC has NTP clients polling the NTP server every second while a program runs – this is unnecessary as the standard NTP implementation

can achieve highly synchronized clocks after only a few exchanges between client and server. Synchronizing clocks, and then achieving a high measure of confidence for the synchronization and then not synchronizing for the duration of a program will reduce the amount of traffic over the network, which will be beneficial for performance-specific traffic.

TOSC is also currently implemented completely in ChucK, so the source files must be manually included in any programs that make use of them. Writing TOSC into the ChucK source code will be beneficial in getting more ChucK users to try the new protocol and will also make TOSC more efficient and faster. Programming practices should also be evaluated – are there performance differences between multicasting and unicasting to each client?

While the protocols implemented here set out to synchronize PLOrk over a network, the mere fact that performers in PLOrk share the same physical space lends itself to imagining various other methods available for synchronization. Aside from relying on people to manually sync to a conductor, one could imagine computers syncing to a sound produced by a server that is inaudible to humans or an electromagnetic wave that is invisible to human eye. While these methods would not have to deal with packet delay, they carry their own issues that must be addressed should they be implemented. Regardless of the methods explored in the future, as the art of interconnected computer music grows, the need to deal with the issues of computers and networks will grow, but it certainly is an issue worth tackling.

Acknowledgements

I would like to thank Perry Cook for his guidance and support throughout this entire project. I would also like to thank Dan Trueman and the rest of PLOrk ’09 for their patience and understanding during rehearsals while I ran my tests.

References

Cáceres, J.-P., Hamilton, R., Iyer, D., Chafe, C., Wang, G., **"**To the Edge with China: Explorations in Network Performance." ARTECH 2008: Proceedings of the 4th International Conference on Digital Arts, pp. 61-66, Porto, Portugal, 2008.

Chafe, C., M. Gurevich, et al. (2004). "Effect of Time Delay on Ensemble Accuracy." Proceedings of the International Symposium on Musical Acoustics.

Chafe, C. and R. Leistikow (2001). "Levels of Temporal Resolution in Sonification of Network Performance." Proceedings of the 2001 International Conference on Auditory Display.

Chafe, C., S. Wilson, et al. (2000). "A Simplified Approach to High Quality Music and Sound Over IP." Proceedings of the COST G-6 Conference on Digital Audio Effects.

Cook, P., P. Davidson, et al. (2005). "Interactive Network Performance: a dream worth dreaming?" Organised Sound **10**(3): 10.

Cook, P., S. Smallwood, et al. (2008). "Composing for Laptop Orchestra." Computer Music Journal **32**: 16.

Dannenberg, R. and P. v. d. Lageweg (2001). "A System Supporting Flexible Distributed Real-Time Music Processing." Proceedings of the 2001 International Computer Music Conference.

Freed, A. (2004). Towards a More Effective OSC Time Tag Scheme, UC Berkeley Center for New Music and Audio Technologies (CNMAT). **Open Sound Control Conference**.

Trueman, D. (2007). "Why a laptop orchestra?" Organised Sound **12**(2): 171-179.

Wang, G. (2006). "ChucK => OSC." Retrieved 03-10-2009, from http://opensoundcontrol.org/implementation/chuck-osc.

Wang, G. (2008). The ChucK Audio Programming Language: A Strongly-timed and On-the-fly Environ/mentality. Computer Science, Princeton University.

Windl, U. and D. Dalton (2006). "The NTP FAQ and HOWTO." from http://www.ntp.org/ntpfaq/.

Wright, M. (2005). "Open Sound Control: an enabling technology for musical networking." Organized Sound **10**(3): 7.

Appendix

AROscSend.ck

```
public class AROscSend {
    3 => int MIN_RATE;
    30 => int MAX_RATE;

    OscSend packets[MAX_RATE];

    int m_portSend;
    int m_portListen;

    Std.getenv("NET_NAME") + ".local" => string myName;

    MIN_RATE => int m_numCopies;
    Std.rand2(1, 255) => int m_seqID;
    0 => int m_lastRateSeq;

    fun void setHost(string hostname, int port) {
        for (0 => int i; i < packets.size(); i++)
            packets[i].setHost(hostname, port);

        port => m_portSend;
        (port + 1) => m_portListen;

        spork ~ listenRateChange();
    }

    fun void startMsg(string msg, string args) {
        for (0 => int i; i < m_numCopies; i++) {
            packets[i].startMsg(msg, "s, i, i, " + args);
            packets[i].addString(myName);
            packets[i].addInt(m_seqID);
            packets[i].addInt(m_numCopies);
        }

        m_seqID++;
    }

    fun void listenRateChange() {
        OscRecv recv;
        m_portListen => recv.port;
        recv.listen();
        recv.event("RateChange, i i") @=> OscEvent oe;

        while (true) {
            oe => now;

            while (oe.nextMsg() != 0) {
                oe.getInt() => int seqID;
                oe.getInt() => int changeFactor;

                if (seqID != m_lastRateSeq) {
                    if (changeFactor == -1 && m_numCopies >= MIN_RATE)
                        m_numCopies--;
                    if (changeFactor == 1 && m_numCopies <= MAX_RATE)
                        m_numCopies++;
```

```
                    seqID => m_lastRateSeq;
                }
            }
        }
    }

    fun void addString(string add) {
        for (0 => int i; i < m_numCopies; i++)
            packets[i].addString(add);
    }

    fun void addInt(int add) {
        for (0 => int i; i < m_numCopies; i++)
            packets[i].addInt(add);
    }

    fun void addFloat(float add) {
        for (0 => int i; i < m_numCopies; i++)
            packets[i].addFloat(add);
    }
}
```

AROscRecv.ck

```
public class AROscRecv {
   5 => int MIN_RATE;
   30 => int MAX_RATE;

   string m_clientList[0];
   int m_lastSeqIDs[0];

   int m_portListen;
   int m_portSend;
   int seenCopies;

   OscRecv recv;
   OscEvent oe;

   fun void port(int portNum) {
       portNum => m_portListen;
       (portNum + 1) => m_portSend;
       portNum => recv.port;
   }

   fun void listen() {
       recv.listen();
   }

   fun void event(string msg) {
       StringTokenizer tok;
       tok.set(msg);

       tok.next() => string msgSeqID;
       " s i i " +=> msgSeqID;

       while(tok.more())
           tok.next() + " " +=> msgSeqID;

       recv.event(msgSeqID) @=> oe;
   }

   fun int nextMsg() {
       0 => int isOnList;

       if (oe.nextMsg() != 0) {
           oe.getString() => string clientName;
           oe.getInt() => int seqID;
           oe.getInt() => int expectedCopies;

           for (0 => int i; i < m_clientList.size(); i++)
               if (clientName == m_clientList[i])
                   1 => isOnList;

           if (isOnList == false) {
               m_clientList << clientName;
               seqID => m_lastSeqIDs[clientName];
               1 => seenCopies;

               return 1;
           }

           if (isOnList == true) {
```

```
                m_lastSeqIDs[clientName] => int lastSeqID;

                if (lastSeqID == seqID) {
                    seenCopies++;
                    return 0;
                }

                if ((lastSeqID + 1) != seqID) {
                    rateChange(clientName, m_portSend, 1);
                }

                seqID => m_lastSeqIDs[clientName];

                if (seenCopies == expectedCopies && expectedCopies > 3) {
                    rateChange(clientName, m_portSend, -1);
                }

                1 => seenCopies;

                return 1;
            }
        }
    }

    fun void rateChange(string client, int port, int changeFactor) {
        OscSend xmit;
        string clientName;

        xmit.setHost(client, port);
        xmit.startMsg("RateChange", "i i");
        Std.rand2(1, 65535) => int randInt;

        for (0 => int i; i < 10; i++) {
            randInt => xmit.addInt;
            changeFactor => xmit.addInt;
        }
    }

    fun int getInt() {
        return oe.getInt();
    }

    fun float getFloat() {
        return oe.getFloat();
    }

    fun string getString() {
        return oe.getString();
    }
}
```

Converting Code from OSC to AROSC

This guide shows basic examples of OSC and the corresponding AROSC implementations. Changing code from using OscSend to AROscSend is as easy as changing a type during declaration. AROscSend uses two ports (one to send and another to listen) – the listening port is automatically set to one plus the value the sending port number is set to, so do not use that port (e.g. if you set 6449 to transmit to the host, do not use port number 6550).

```
OscSend xmit;
xmit.setHost("localhost", 6449);

while( true )  {
    xmit.startMsg( "/msg", "i f s" );
    1 => xmit.addInt;
    1.0 => xmit.addFloat;
    "hi" => xmit.addString;

    0.2::second => now;
}
```

```
AROscSend xmit;
xmit.setHost("localhost", 6449);

while( true )  {
    xmit.startMsg( "/msg", "i f s" );
    1 => xmit.addInt;
    1.0 => xmit.addFloat;
    "hi" => xmit.addString;

    0.2::second => now;
}
```

Changing code from using OscRecv to AROscRecv requires changing the declaration type from OscRecv to AROscRecv. In AROscRecv, the OscEvent object used in OscRecv is integrated directly into AROscRecv objects so calls to the event() method and subsequent waiting and parsing packets is done on the AROScRecv object, not an OscEvent object.

```
OscRecv recv;
6449 => recv.port;
recv.listen();
recv.event( "/msg, i f s" ) @=> OscEvent oe;

while ( true )  {
    oe => now;

    while ( oe.nextMsg() != 0 )  {
        oe.getInt => int myInt;
        oe.getFloat => float myFloat;
        oe.getString => string myString;

    }
}
```

```
AROscRecv recv;
6449 => recv.port;
recv.listen();
recv.event("/msg, i f s");

while ( true )  {
     recv.oe => now;

     while (recv.nextMsg() != 0)  {
       recv.getInt => int myInt;
       recv.getFloat => float myFloat;
      recv.getString => string myString;
     }
}
```

NTPServer.ck

```
public class NTPServer
{
    int port_timeRequest, port_timeReplies, port_RTTinfo, port_RTTbroadcast;

    0 => int RTTreported;
    0.0 => float RTTsum;

    1000.0 => float TTO;

    OscRecv recv;
    OscEvent oe;

    OscSend TTOsend;

    fun void port(int portNum) {
        portNum => port_timeRequest;
        (portNum + 1) => port_timeReplies;
        (portNum + 2) => port_RTTinfo;
        (portNum + 3) => port_RTTbroadcast;

        TTOsend.setHost("224.0.0.1", port_RTTbroadcast);

        portNum => recv.port;

        spork ~ listen();
        spork ~ RTTlisten();
        spork ~ broadcastTTO();
    }

    fun void listen() {
        recv.listen();
        recv.event("/ntp/request, s f") @=> oe;

        OscSend send;

        while (true)  {
            oe => now;

            while(oe.nextMsg() != 0)  {
                oe.getString() => string host;
                oe.getFloat() => float clientTime;

                send.setHost(host, port_timeReplies);
                send.startMsg("/ntp/reply", "s f f");

                host => send.addString;
                clientTime => send.addFloat;
                now / samp => send.addFloat;
            }
        }
    }

    fun void RTTlisten() {
        OscRecv recv;
        port_RTTinfo => recv.port;
        recv.listen();
```

```
        recv.event("/ntp/rtt, s f") @=> OscEvent roe;

        while (true) {
            roe => now;

            while (roe.nextMsg() != 0) {
                roe.getString() => string host;
                roe.getFloat() => float RTT;

                RTT + RTTsum => RTTsum;
                1 +=> RTTreported;

                if (RTTsum/RTTreported < .2 * TTO && RTTreported > 3) {
                    Math.round(TTO * 0.5) => TTO;
                    broadcastTTO();
                }

                if (RTTsum/RTTreported > .6 * TTO && RTTreported > 3) {
                    Math.round(TTO * 2.0) => TTO;
                    broadcastTTO();
                }
            }
        }
    }

    fun void broadcastTTO() {
        while (true) {
            TTOsend.startMsg("/ntp/tto", "f");
            TTOsend.addFloat(TTO);

            10::second => now;
        }
    }

    fun time currentTime() {
        return now;
    }

    fun float getTTO() {
        return TTO;
    }

    fun float avgRTT() {
        return RTTsum/RTTreported;
    }
}
```

NTPClient.ck

```
public class NTPClient
{
    Std.getenv("NET_NAME") + ".local" => string myName;

    int port_timeRequest, port_timeReplies, port_RTTinfo, port_RTTget;

    1000.0 => float TimeToExecute;

    OscRecv recv;
    OscEvent oe;

    OscSend send;

    float RTT, offset;

    string host;

    fun void setHost(string hostname, int port) {
        send.setHost(hostname, port);

        port_timeReplies => recv.port;

        hostname => host;

        port => port_timeRequest;
        (port + 1) => port_timeReplies;
        (port + 2) => port_RTTinfo;
        (port + 3) => port_RTTget;

        spork ~ requestTime();
        spork ~ getTime();
        spork ~ getRTT();
    }

    fun void requestTime() {
        while(true) {
            send.startMsg("/ntp/request", "s f");
            send.addString(myName);
            send.addFloat(now/samp);

            1::second => now;
        }
    }

    fun void getTime() {
        OscRecv timeGetter;
        port_timeReplies => timeGetter.port;
        timeGetter.listen();
        timeGetter.event("/ntp/reply, s f f") @=> OscEvent toe;

        OscSend RTTdata;
        RTTdata.setHost(host, port_RTTinfo);

        while(true) {
            toe => now;

            while (toe.nextMsg() != 0) {
                now / samp => float timeReplyReceived;
```

```
                toe.getString() => string host;
                toe.getFloat() => float timeRequestSent;
                toe.getFloat() => float serverTime;

                timeReplyReceived - timeRequestSent => RTT;
                serverTime - timeRequestSent => offset;

                RTTdata.startMsg("/ntp/rtt", "s f");
                RTTdata.addString(myName);
                RTTdata.addFloat(RTT);
            }
        }
    }

    fun void getRTT() {
        OscRecv RTTget;
        port_RTTget => RTTget.port;
        RTTget.listen();
        RTTget.event("/ntp/tto, f") @=> OscEvent roe;

        while(true) {
            roe => now;

            while (roe.nextMsg() != 0)
                roe.getFloat() => TimeToExecute;
        }
    }

    fun time currentTime() {
        return (now + offset*samp + .5 * RTT*samp);
    }

    fun float getTTO() {
       return TimeToExecute;
    }
}
```

TOscSend.ck

```
public class TOscSend
{
    Std.getenv("NET_NAME") + ".local" => string myName;

    OscSend send;

    int portSend;

    string host;

    fun void setHost(string hostname, int port) {
        send.setHost(hostname, port);

        hostname => host;
        port => portSend;
    }

    fun void startMsg(string msg, string args, float currentTime, float TTO) {
        send.startMsg(msg, "s, f, f, " + args);
        send.addString(myName);
        send.addFloat(currentTime);
        send.addFloat(TTO);
    }

    fun void addString(string add) {
        send.addString(add);
    }

    fun void addFloat(float add) {
        send.addFloat(add);
    }

    fun void addInt(int add) {
        send.addInt(add);
    }
}
```

TOscRecv.ck

```
public class TOscRecv
{
    int portListen;

    OscRecv recv;
    OscEvent oe;

    fun void port(int portNum) {
        portNum => portListen;
        portNum => recv.port;
    }

    fun void listen() {
        recv.listen();
    }

    fun void event(string msg) {
       StringTokenizer tok;
       tok.set(msg);

       tok.next() => string msgSeqID;
       " s f f " +=> msgSeqID;

       while(tok.more())
           tok.next() + " " +=> msgSeqID;

       recv.event(msgSeqID) @=> oe;
    }

    fun float nextMsg(float myTime) {
        if (oe.nextMsg() != 0) {
            oe.getString() => string host;
            oe.getFloat() => float currentTime;
            oe.getFloat() => float TTO;

            return (currentTime + TTO - myTime);
        }

        else
            return 0.0;
    }

    fun int getInt() {
        return oe.getInt();
    }

    fun float getFloat() {
        return oe.getFloat();
    }

    fun string getString() {
        return oe.getString();
    }
}
```