Portfolio of Research

by

John Paul Young

Submitted in partial fulfillment of the
requirements for the degree of
Master of Music
in Computer Music—Research
at the Peabody Conservatory of Music
The Peabody Institute of The Johns Hopkins University
Baltimore, Maryland

April 2003

CONTENTS

ACKNOWLEDGEMENTS

This portfolio would not have been possible without the inspiration and wisdom of McGregor Boyle, Ichiro Fujinaga, and Geoffrey Wright. Their decades of experience in connection with computer music were essential to the inception and progress of my research. Dr. Fujinaga, as professor, mentor, and friend, deserves particular recognition for helping me to stay focused, challenging me to learn what I thought I already knew, and introducing me to our wonderful community. He will always be *sensei*. I also owe particular gratitude to my friend and collaborator Randall Packer, whose wonderfully impractical ideas pushed me beyond the limits I would have otherwise set for myself. Countless other cherished colleagues around the world have opened my eyes, ears, and mind to ways of approaching music that have enhanced my understanding and appreciation for this art form that we cannot live without. NOVA Research Company provided me with the infrastructure to carry out much of my research and development, and my coworkers demonstrated admirable forbearance when I was often lured from my post as systems administrator by the irresistible charms of music. I cannot express sufficient thanks to my family for supporting my studies in innumerable ways—they may not fully understand my quest, but the fact that it makes me happy is all the justification they need. Finally, very special acknowledgement must be given to my partner and *sempai* Margaret Schedel. She provided invaluable assistance in developing, testing, and presenting portions of my work. More importantly, as an accomplished computer musician in her own right, she has been a constant source of thoughtful feedback, both keeping me grounded and encouraging me to leap into the unknown.

Artists long for limitations; excessive freedom casts us into a vacuum.
Style used to be an interaction between the human soul and tools that were limiting.
In the digital era, it will have to come from the soul alone.

— Jaron Lanier

INTRODUCTION

This portfolio brings together in chronological order the research I performed while obtaining my Master of Music degree at the Peabody Conservatory of Music. Each chapter represents a project consisting of both ideas and implementations. Though the projects as documented here were necessarily preserved at their time of completion, the vision manifested in each case extends beyond these pages into the present and hopefully into the future. The arc of technological obsolescence is brutal, so some of the details herein already betray their age. Nevertheless, if other explorers encounter these cooling embers and can use them to illuminate their own investigations, then I will have reason to be proud of this work.

I have been fortunate to receive generous acknowledgement and support of my research from the computer music community throughout my course of study, with opportunities to present my findings at conferences around the world. *Piano Master Classes via the Internet* was selected as a poster for the 1999 International Computer Music Conference (ICMC) in Beijing, China. *Using the Web for Live Interactive Music* was chosen as a paper for the 2001 ICMC in Havana, Cuba. A version of the same project was also selected as a paper for the "Music Without Walls? Music Without Instruments?" conference at De Montfort University, Leicester, UK in 2001. It was subsequently published under the title *Networked Music: Bridging Real and Virtual Space* in *Organised Sound* 6(2) by Cambridge University Press. Finally, *Application of Fault Tolerance to Interactive Music* was chosen for presentation at the 2001 Society for

Electro-Acoustic Music in the U. S. (SEAMUS) conference in Baton Rouge, LA, and published in *Journal SEAMUS* 16(1).

I have also been privileged to serve on the board of the International Computer Music Association (ICMA) as Publications Coordinator. This position has granted me valuable perspective on the direction of the field as it is driven forward through the cultural bazaar of musical thought and practice by the composers, inventors, dreamers, and pranksters that make up the computer music community. Additionally, through a collaboration between the ICMA and *Organised Sound*, I am currently co-editing issue 8(3) of the publication. This experience has not only allowed me to learn the mechanics of producing an international journal, but through the process of reviewing and editing the research and writing of my peers, has given me further insight into improving the quality of my own endeavors.

This portfolio reflects my chosen emphasis of research in pursuit of the Master's degree. However, I consider myself to be both composer and performer as well. During my time at Peabody, I have been commissioned to write several film scores for the NASA Scientific Visualization Studio at the Goddard Space Flight Center, in addition to performing my own compositions at SEAMUS 2002 in Iowa City, IA, and New Interfaces for Musical Expression (NIME) 2003 in Montreal, Canada. I have tried to leverage this breadth of ambition and background towards an integrative approach to the issues surrounding computer music, not only for my own benefit, but in order to build bridges of understanding within the community. I believe this effort is apparent in the

projects I have chosen, emphasizing the benefits of collaboration, and recognizing the various roles and relationships essential to the advancement of music as both personal and public phenomenon. Studying these unquantifiable topics is the realm of philosophers, which we seem to have a shortage of in contemporary music academia. I do not presume to fill this void, yet I have striven to reconcile the sometimes conflicting viewpoints of art, science, and culture by seeking out common ground in the space of my own imagination and in discourse with colleagues. As a researcher, new tools enable new music, and better tools lead to better music. As a composer, the audience can be either embraced or denied, but its presence informs every decision regardless. As a performer, music imparts magical powers, the ability to speak a universal language of the spirit. As an audience member, nothing matters but the portal opening to another dimension for as long as the music lasts. Each of these facets within myself informs the others, and each focus is but an attempt to reach greater intimacy with the primal musical experience via a different path. What the reader can absorb from these pages is only one aspect of this challenge, a step on the journey to find the meaning of music.

At first glance, the three projects described here seem to have little connection to each other. The surface thread that binds them is the typical researcher's effort to break down existing limitations. In the first chapter, the barrier is geography; in the second chapter, traditional concepts of musical interaction; in the third chapter, dependability of computer instruments. I have attempted to circumvent these limitations through creative application of technology: internetworking, the World Wide Web, and fault tolerance techniques. In accordance with the disparate disciplines unified in computer music, I have brought to

bear not only my training as a musician, but roughly fifteen years of employment as a computer technician, network engineer, and multimedia developer. So, the deeper current flowing through these projects is fed by both artistic motivation and practical realism: a desire to transform the ways in which music can be experienced, in a democratizing fashion, without sacrificing standards of originality and expertise. I don't claim to have invented this goal—it is shared by many others in the academic and popular arenas with more profound credentials than mine. I merely aspire to add my own contribution to maintaining music as an ongoing collective venture of humanity rather than an artifact to be preserved or marketed. In some respects, music must compete with all the other demands on the finite course of a lifetime—there are countless ways to work and play. In other respects, music is like light—it seems to occupy no space, it can make nearly any environment more vibrant, and while we can technically live without it, its absence would make for an impoverished life, indeed. Perhaps another way of framing the impulse behind these projects is simply as a wish to share the joy that music brings into my own life. I expect every musician feels the same inclination, and I believe acting on this impulse is what will keep music evolving and thriving into the future. In my own way, I have tried to expand notions of the possible by lending energy to this pursuit, adding my voice to the chorus, and infusing my character into the whole.

Beyond the conceptual framework binding these projects together, I also had an opportunity to weave them into a technological fabric in support of an intermedia work dubbed *Netaphor*. In the spring of 2001, a new experimental workshop was offered as a partnership between the Peabody Conservatory, the Maryland Institute College of Art

(MICA), and the Johns Hopkins University (JHU). The class was taught by a team of professors—McGregor Boyle, Geoffrey Wright, and Randall Packer—with the objective of bringing students from different media backgrounds together to work on collaborative projects. The final goal of the workshop was to showcase completed works at the public inauguration of the JHU Digital Media Center (DMC). Margaret Schedel and I served as artistic and technical advisors for the course, as well as crafting a project of our own in collaboration with Kari Heath, Fawad Kahn, and Galo Moncayo. The objective of *Netaphor* was to sample real-time audio and video input from multiple locations during the DMC event, then mix and manipulate the sources digitally. The processed sound and imagery were rebroadcast into a room designed as a 'telematic time capsule'—a live distillation of the sensory experience of all the other projects, a vessel of intermedia improvisation. To fulfill this goal, I had to rely on everything I learned and developed during the course of my research. A mixing board was transformed into a virtuosic intermedia instrument, using MIDI data distributed via the network to control operation of the entire system from a single console. Audio and video were streamed into this command center, seamlessly blended with other live and prerecorded sources, and streamed out to the time capsule over the network, blurring boundaries between the real and the virtual. Margaret and I pushed the software to the limits of its capabilities, resulting in a level of instability that was mitigated by substantial redundancy and rapid failover capability.

While *Netaphor* was a mixed success overall for reasons unrelated to its concept or design, the technical underpinnings of the work functioned remarkably well. It was tremendously gratifying to be able to apply the sum of my research towards an artistic goal and find concrete value beyond its original context. The end result wasn't exactly music, but something from the border zone between reality and imagination, a transformation of the tangible into the ephemeral, a window into somewhere inexplicable just out of reach. So it wasn't entirely not music, either.

Piano Master Classes via the Internet

The key to the mystery of a great artist is that, for reasons unknown,
he will give away his energies and his life
just to make sure that one note follows another inevitably…
and leaves us with the feeling
that something is right in the world.

— Leonard Bernstein

1.1 Research Objective

The Internet has made many new ways of experiencing music possible. Considerable resources are currently being applied to the problems of delivering sound over computer networks using a variety of different techniques. Some possibilities have remained largely unexamined, prompting the focus of this research. The objective of this project is to demonstrate that transmitting live performance between properly equipped acoustic pianos using the Musical Instrument Digital Interface (MIDI) protocol over the Internet is feasible, and in fact audibly superior to other existing means of reproducing remote piano performance. Achievement of this objective would provide the fidelity necessary to support piano Master Classes as a realistic option for conservatory-level distance education.

1.2 MIDI and the Piano

The MIDI protocol (hereafter referred to simply as MIDI) is uniquely suited to faithful capture and reproduction of piano performance. MIDI was developed with an orientation towards the keyboard-based synthesizers prevalent at the time, so it is particularly effective at accurately describing the gestures possible on such an instrument (Chadabe 1997, 196). There is not universal agreement on this point, but the sheer ubiquity of MIDI has overwhelmed most critics and made it very useful despite imperfections (Loy 1985). Also, because most traditional piano music does not require a performer to directly access the strings and soundboard, the interface of the keys and pedals themselves limits the

range of possible actions. By converting those actions into numerical shorthand, MIDI is theoretically capable of representing any keyboard gesture that can be written in standard notation. Sending MIDI data to a piano equipped to interpret and play back these representations can result in an exceptionally realistic reproduction of a performance.

A MIDI-enabled acoustic piano has the same external characteristics as an ordinary acoustic piano, both sonically and functionally. A MIDI piano is distinguishable mainly by a component box mounted on the underside of the instrument, usually including a display and controls accessible from the playing position. The digital interface interprets data from sensors beneath the keyboard action mechanism that measure which key is played, how quickly it is depressed, when it is released, and how quickly it is released, in addition to pedal positions. These parameters are converted into MIDI messages that can be sent to any device capable of processing MIDI data, such as a synthesizer or computer. A fully MIDI-enabled piano also includes an apparatus that can trigger the keys, hammers, and pedals according to received MIDI parameters, replicating the mechanics of the original performance. Currently there are both manufacturer-integrated MIDI-enabled pianos such as the Yamaha Disklavier, and similar mechanisms that can be retrofit to standard acoustic pianos, such as those sold by PianoDisc. For this project, we intend to use a computer to relay MIDI messages from one computer-connected piano over a network to another comparable piano at a separate location.

1.3 Master Classes and Distance Learning

Transmitting MIDI over a network could be useful in a variety of distributed musical interactions. The goal of this project is a minimum-compromise solution, which would be particularly appropriate to professional performance, audiophile-quality broadcasting, and conservatory-level instruction. Due to the delay inherent in long-distance communication, this goal could only be achieved for bidirectional situations by limiting usage to non-simultaneous interaction. There might be other potential applications in the area of musical distance education beyond the scope of this paper, but in light of the superior quality and not-quite-real-time nature of this system, it seems well-adapted for Master Classes. A Master Class is a special instructional environment in which a virtuoso musician demonstrates techniques to a select group of students, observes students' skills, and offers feedback for improvement. Traditionally it has been a hands-on atmosphere, in which the Master carefully notes physical elements of technique such as posture, hand positions, and other aspects of body control as well as audible elements like accuracy, tone, and interpretation. Clearly, a distance Master Class would not be entirely comparable to the classic model, but could compensate by addressing a much wider audience. The possibility of sharing even a significant fraction of a Master's expertise would offer a compelling experience to many students unable to participate otherwise.

Of course, a Master Class involves more than just musical information; it also requires interaction between teacher and student(s). Our expectation is that current and future videoconferencing technology will be sufficient for this purpose. Inexpensive cameras

and free software are already used for many types of distance education, and the quality and affordability of these solutions should only continue to improve.

1.4 Internet Communications

The Internet was not constructed for the purpose of transmitting audio or video. It was built as a distributed framework of peers optimized so that failure at any one or even multiple points would not severely impact the overall functionality of the network (Comer 1995, 49). Unlike the telephone system, in which a persistent circuit link is created between two endpoints whenever a call is placed, Internet connections are largely virtual. Nodes such as personal computers communicate via sessions, fragmented into small discrete packets tagged with their origin and destination addresses, over an underlying architecture with no knowledge of the relationship of one packet to the next. Even continuous data such as audio is divided into these packets, which must be transparently reconstructed at the receiver into their original relationship in order to play back properly. The functioning of this packet-based architecture depends primarily on routers—specialized computers that analyze each incoming packet, discern its destination, and forward it to another router believed to be closer to that destination, until eventually the packet reaches its target (Comer 1995, 109). Each packet finds its own path through the router network, which can potentially be much longer or shorter than that of the packet just before or after it. Routers do not conform to fixed pathways, but rather monitor traffic flow and use knowledge of other nearby routers to dynamically optimize packet transit in response to changing conditions. Between any two nodes there

are probably several and often more than twenty routers, or "hops", through which each packet of data must pass. Due to the finite speeds of electrons in a wire and light in a fiber-optic cable, there is a physically imposed limit on how fast these packets can travel, introducing a minimum delay based on the distance traversed (Eliens, et al. 1997). Also due to the design of the Internet, this distance is always greater than the direct geographical distance between nodes, sometimes significantly so. Thus it is appropriate to imagine communication between two computers over the Internet not as an orderly stream of information, but as a swarm of small shepherded particles that intelligently reassemble upon their arrival. Of course, this process doesn't always happen the way it should. Routers can become overwhelmed by heavy data traffic. When they fall behind in processing, packets get placed into a finite memory buffer, first adding extra delay to those pieces of data. If the buffer space runs out—which will eventually happen unless traffic declines—new incoming packets must be ignored, and thus never reach their destination at all. These characteristics of the Internet have direct implications for the success of live music transmission.

1.5 Theoretical Obstacles: Packet Loss, Latency, and Jitter

There are two broad potential problem areas in reproducing live performance using MIDI over the Internet. First, there are possible limitations of MIDI itself as a means of capturing an accurate and comprehensive signature of a piano performance. That topic will not be addressed in depth here, as it has been thoroughly debated elsewhere in the literature (Moore 1988). In the course of this research, testing has determined that MIDI

is the least limiting factor in real-time remote reproduction quality. Second, and more prominently, there are inherent issues of transferring real-time media over the Internet that must be addressed. These issues are data loss, latency, and jitter. Data loss is simply when information sent does not reach the receiver. Latency is the inherent transmission delay between any two physically distant points, with a minimum determined by the speed of light, and no absolute maximum. Jitter is the variation in latency between different messages, primarily of note when it causes information to be received in a different order than it was sent.

Intermittent data loss is generally caused by excessive traffic, which overloads routers and results in denial of service through those routers until traffic is reduced. Usually this downtime is very brief, but is difficult to anticipate and avoid because it results from erratic spikes in traffic intensity. Similarly, the latency between any two nodes varies from one moment to the next, sometimes significantly so, causing jitter and raising the possibility that musical data will arrive after it was supposed to be played. With millions of simultaneous users, the Internet has become much like the Earth's weather: localized behavior is almost impossible to predict with certainty, so being prepared for anything is the best defense against undesirable outcomes.

1.6 Practical Differences between MIDI and Audio

Current schools of thought in transmitting musical information have converged towards two options: using relatively high bandwidth to send digitized audio samples, or using

relatively low bandwidth to send musical representations (Casey and Smaragdis 1994). In a typical sampling approach the source would be recorded with a microphone and pass through an analog-to-digital converter (ADC), generating 44,100 samples per second for CD-quality sound. The samples would be sent over the network and upon arrival pass through a digital-to-analog converter (DAC), reconstructing the audio for output by an amplified loudspeaker. Assuming a high-quality ADC and DAC, the distortion introduced by the digitization process is acceptable, but there are significant compromises in reproducing piano music using microphones and loudspeakers (Miller 1996). Also, the bandwidth consumed by this process is 1.346 Mbit/sec for a stereo signal, enough to nearly saturate a 1.544 Mbit/sec-capacity T1 network connection, a typical standard of high bandwidth. Accommodating such data rates would tax the network infrastructure of almost any institution.

Musical representations in the most general sense, of which MIDI is the most common digital example, transmit pre-processed abstractions, which are reproduced by whatever synthesis device is available at the receiving station. These abstractions are compact and make efficient use of limited bandwidth. There are two clear potential compromises to the abstraction approach. First, unless the reproduction platform is exactly the same as that used for recording, the result will not be timbrally faithful to the source. Given the immense variety of hardware and software synthesizers in use, configuring matched sending and receiving systems can be difficult, but use of comparable MIDI-enabled pianos can circumvent this issue. Second, because each abstraction represents some larger musical construct, loss of a single abstraction is likely to have more perceptual

significance than loss of a single sample. For example, loss of a packet containing a MIDI note-on message could result in several seconds of obvious musical inaccuracy, whereas loss of a packet containing one CD-quality sample would produce imprecision lasting only 1/44,100th of a second, a barely noticeable deviation. Thus the efficiency of abstractions can be offset by a greater sensitivity to error.

MIDI is a very efficient format, requiring very little network bandwidth relative to audio samples. By reducing most musical gestures to transition states, MIDI can describe most of the range of piano performance in simple three-numeral messages. For example, playing middle C at *forte* might trigger the message "144 60 75", representing a note-related instruction, the note C4, at a velocity of 75, respectively. One second later, releasing the note might transmit "144 60 0", the velocity of 0 indicating that the note has been released. Thus in 6 bytes, a musical action has been captured in roughly 1/30,000th of the size the equivalent audio would have used. Of course, most real music is more dense than this example, but even an entire pyrotechnic work such as the Liszt Piano Sonata in B Minor is compressed to roughly 1/2000th using MIDI, in arguably lossless form. The complete MIDI specification describes the range of expression possible using these straightforward messages (MMA 1996).

With MIDI-enabled acoustic pianos used as both sender and receiver, the sound quality exceeds any other means of reproduction. Regardless of the quality of live audio streaming, no arrangement of microphones and speakers can match the acoustics of a piano as accurately as hammers striking the strings of a real, physical instrument (Miller

1996). MIDI-enabled pianos have an increasing presence in the musical community, as their multipurpose flexibility is recognized and their price premium diminishes. By connecting such pianos to a computer with Internet access, one can broadcast a performance or tune in to someone else's, using the software developed over the course of this project.

1.7 Distinctions between the UDP and TCP Protocols

Several issues were encountered during design and development of the network software component of the project. First came the consideration of protocols. The only choice of network protocol was IP (Internet Protocol), the standard protocol of the Internet. There remained the decision between TCP (Transport Control Protocol) and UDP (User Datagram Protocol), both of which interoperate with IP for transport.

TCP is designed for reliability, to minimize data transmission errors by maintaining a constant dialogue between sender and receiver, acknowledging receipt of packets and adjusting for variable network conditions. As a result, TCP functions poorly in time-sensitive applications over long distances with many routing hops. There are basically three problems with TCP from a musical viewpoint: (1) To guarantee delivery of all data, TCP retransmits lost packets, causing music to stop while dropped events are redelivered. (2) TCP ensures that packets arrive in the same order they were sent, causing notes to back up waiting for the arrival of differently routed notes. (3) TCP includes slight extra

bandwidth and processing overhead to perform these error corrections (Comer 1995, 191).

To avoid these problems, UDP was chosen for packet transmission. UDP is an "unreliable" protocol, with nominal overhead, but no inherent compensation for dropped and out-of-order packets. UDP is also completely connectionless, meaning there is no inherent dialogue between sender and receiver, so UDP has no provisions for flow control or response to changing network conditions. The sender transmits packets as fast as it can, and the receiver must be equipped to process them as they arrive. Thus, UDP forces the higher-level application to deal with network issues when they occur. It should be noted that UDP, like TCP, incorporates a packet-level checksum verification to ensure data integrity, thus sparing the application from this necessity (Comer 1995, 179).

1.8 Comparison of the MAX/OSC and Java Environments

The initial plan for this project was to exploit Java's networking and user interface strengths in combination with its touted cross-platform compatibility to develop a solution that would function for the maximum number of possible users. However, even with current releases of the Java Media Framework and JavaSound programming interfaces (Sun 1999), there are no standard Java classes for processing MIDI device input. Third-party implementations of MIDI I/O exist, but there also seems to be a consensus—expressed on computer music newsgroups—that the current Java VM (Virtual Machine) is not suitable for time-sensitive event processing because of operating

system (OS) latency and timing instability. Testing confirmed these constraints. General degradation in OS stability was also observed, presumably introduced by the need to constantly exchange data between Java code and Native Interface drivers compiled in C. So, cross-platform compatibility was abandoned for this testing phase, and Opcode's MAX programming environment (Zicarelli 1998) on MacOS was chosen instead. MAX has excellent support for MIDI processing, but conspicuously lacks any built-in networking objects. Fortunately, in support of their Open Sound Control (OSC) initiative, the Center for New Music and Audio Technologies (CNMAT) at the University of California, Berkeley has made available exactly what was needed—OTUDP (Open Transport UDP), a MAX object that transmits data over IP using UDP (Wright 1998). These tools enabled quick prototyping of a system for demonstrating feasibility and functionality of the software.

1.9 Related Existing Initiatives

There is existing software that attempts to solve some of the issues discussed above, but none have achieved the level of fidelity and responsiveness necessary to support an environment such as a long-distance Master Class. MIDIShare, developed by Grame Computer Music Research Laboratory, facilitates real-time MIDI communications over a local-area network (LAN), but does not implement the necessary error-correction to successfully operate over the Internet (Fober 1994). Popular streaming-audio solutions use special network server hardware to compress audio data either live or in advance and facilitate reliable transmission. Such products include RealAudio, QuickTime,

Shockwave, and various implementations based on the MPEG-Layer 3 (MP3) standard. However, by applying heavy data compression, audio quality is sacrificed for the sake of reduced bandwidth.

One attempt to improve the fidelity of abstracted representations is Beatnik (Beatnik 1999), which essentially sends the same synthesized "patches" used at the source ahead of the musical data, to be used for reproduction by the receiver. Another significant step in this direction is the Structured Audio Orchestra Language (SAOL), a component of the MPEG-Layer 4 standard, which transmits unit-generator-based synthesis instructions similar to a programming language, much like Csound (Scheirer 1998). While such approaches promise increased fidelity for pre-composed material, there is still no way to distill live performance into these formats.

1.10 Solutions: Redundancy, Indexing, and Buffering

We devised solutions to mitigate each of the technical hurdles facing a real-time MIDI broadcast over the Internet in order to achieve acceptable live performance reproduction. To reduce data loss, multiple copies of each MIDI message are sent, to increase the probability that at least one copy will reach the receiver (Ramanathan 1992). To mitigate latency, a receiving buffer proportional to the average delay is established that allows messages to "catch up" with those ahead of them for seamless playback. Finally, to compensate for jitter, outgoing messages are tagged with an index number and time

stamp, which allows them to be reassembled in the proper sequence and rhythm by the receiver.

Sending multiple simultaneous copies of each MIDI message increases the odds that one of the copies will successfully arrive at its destination, but one complication to this approach is that if a router becomes overwhelmed and is forced to drop packets, it will lose all those temporally near to each other until it recovers. Packets in a tight cluster are far more likely to all experience the same conditions than if they were widely spaced out in time, so even though multiple copies theoretically increase the likelihood of arrival in linear proportion to the number of instances, this router behavior diminishes those odds by an unknown—and unknowable—factor (Bolot 1993). To compensate for this aspect of the Internet architecture, two additional steps could be taken. A pre-broadcast "line test" could be implemented to determine nominal packet loss rates in order to set an appropriate level of redundancy for current conditions. Also, in addition to the stream of MIDI data traveling from sender to receiver, a small amount of diagnostic information could be sent back to the transmitting station, to allow for constant monitoring of network conditions and adjustment of operating parameters. By comparing the number of events received to the number sent, network packet loss rates could be approximated, and redundancy could be optimized without interrupting performance. These remain imperfect solutions, as it was noted previously that conditions on the Internet can change more rapidly and substantially than can be addressed in time to compensate. Fortunately, in practice, simple fixed redundancy resulted in virtual elimination of dropped MIDI events in both local and transnational tests.

The next problem, latency, was dealt with by adding a receiving buffer proportional to the packet travel time between origin and destination nodes. This buffer allows packets that follow different paths through the Internet to reconvene in the proper order before being output. This technique introduces a pause between when performance information is sent and when it is reproduced by the receiver, but such a compromise is necessary to prevent the stuttering playback that would otherwise likely occur. At this time, the size of the buffer is set manually, by trial-and-error. It could also be determined initially by the "line test" described above, which in addition to packet loss rates could easily measure round-trip delays between nodes over several seconds' time, recording minimum, maximum, and mean latencies in order to determine optimal initial buffer length. Monitoring diagnostics could enable an increase in buffer size to cope with changing network conditions. By measuring a periodic round-trip "ping", network latency levels could be calculated, and buffer size adjusted accordingly. Buffer size can not be decreased on-the-fly without potentially disrupting data already in the queue, but it can be manually adjusted at any pause in the transmission if so desired.

The final problem, jitter, was handled by indexing each MIDI event sequentially and measuring the time between events in order to ensure proper playback order and rhythm. In terms of observed aggregate behavior, the Internet has a tendency to transmit in bursts, so packets sent out over a span of time will likely be temporally clustered, arriving more closely packed together, in addition to potential reordering (Clark and Lehr 1999). It would be musically unacceptable for a performance to be altered in such a manner.

Tagging each MIDI event with an index and time stamp provides all the information necessary to reconstruct the exact order and rhythm of the original event stream. These measures completely eliminate jitter as a factor in reproduction fidelity, assuming a buffer size large enough to cope with the demonstrated range of latencies.

To enhance these error-correction schemes, we have also experimented with algorithms to gracefully release stuck notes if the expected note-off message does not arrive, but these extensions have not been fully tested. This problem is less severe in a piano-based system than it might be otherwise, due to the natural decay of any held note into silence.

1.11 Time Tracking: Delta vs. Running Time Stamps

We discovered that one of the most important factors in mitigating the problems of live Internet broadcast was the use of time-stamping. However, there were two distinct potential implementations, and the superior solution could not be objectively determined. These two approaches were delta-time and running-time. Delta-time is a relative measurement, recording only the time since the previous event occurred. Running-time is an absolute measurement, recording the elapsed time since the beginning of some reference point. Delta-times are more efficient, in that they will typically be numerically smaller than running-times. Since these stamps are attached to each event, the difference in data overhead could be significant. However, in the case of dropped events, with no awareness of a lost event's duration, a delta-time system will "skip" to the next event as if the dropped event had never existed. By contrast, a running-time system will recognize

that a dropout occurred, respect the time it would have occupied, and maintain the original relationship of the events that were successfully transmitted. In some contexts, the delta-time behavior would be more desirable, but in a musical context, a "skip" is far more disruptive to the listening experience than a silent absence that maintains the correct underlying rhythm. With the decision thus made in favor of running-time, a reference point for measurement had to be chosen. Simply starting a millisecond-resolution timer at the beginning of a session soon resulted in stamps so large the system slowed down just processing the data. To fix this, an additional feature was developed to reset the running-time to zero after a certain amount of time passing without any activity, resetting the index value to zero as well. This window is manually adjustable, and optimal parameters can vary based on the style of music being played and responsiveness desired. Due to peculiarities of the MAX timing subsystem, the running-time finally implemented is actually computed by accumulating delta-times, with the result equivalent to an independent clock, but more accurate and efficient. Also, though the clock in MAX appears capable of 1 ms resolution, we found 5 ms to be the minimum measured differential between events, which was determined to be both musically acceptable and sufficient to precisely regenerate MIDI data.

1.12 Operation of the Experimental System

Once a platform and environment were chosen, implementing the logic of our system was relatively straightforward. To get a sense of the entire process from playing a note on the sending piano to hearing it reproduced on the receiving piano, we'll follow an event

through the flow of execution. When a note is originally struck, its MIDI event data is passed to MAX. The running-time is computed by adding the delta-time since the last event or clock reset, and this time stamp is bundled with the new event. This bundle is subsequently attached to a numeric index that increments by one for each event. Note and controller messages are both treated the same. Here's what our hypothetical first note might look like:

Index	ms	| MIDI msg |	
1	44	176 64 127	(Controller 64—Sustain, with value 127—Full)
2	59	144 60 98	(Note 60—C4, with velocity 98—Forte)
3	559	144 60 0	(Note 60—C4, with velocity 0—Off)
4	688	176 64 0	(Controller 64—Sustain, with value 0—Off)

Each bundle is next fed through a duplication loop based on the current level of redundancy. The multiple bundles are then sent across the network to the receiving station. The received data is placed in a database sorted by the index value, such that duplicates merely overwrite the same location, a slight but acceptable inefficiency. The receiver clock is started by the arrival of the first event or a clock reset message, and counts until the current buffer length is reached. After the buffer duration has elapsed, each bundle is triggered to be sent out from the database in numerical order at the specified time value, and sent to MIDI output, at which point it is reproduced on the target piano. After each event has been played, its index is fed back into a low-priority pipeline that erases the event from the database, preventing excessive memory allocation.

1.13 General Internet Performance Metrics

We conceived of other refinements to the system, as there are many parameters that can be independently modified. For example, OTUDP itself allows adjustment of the size and number of internal UDP buffers for both outgoing and incoming data. Aligning the UDP buffer size with the underlying network MTU (Maximum Transfer Unit) size to prevent packet splitting could theoretically increase efficiency and performance, although in practice this is difficult over a heterogeneous network where MTUs may vary (Comer 1995, 95). Without access to the source code, optimum settings for OTUDP have been determined through testing, and for this application the minimum buffer size of 128 bytes and buffer quantity of 2048 units seemed to work quite well in both LAN and Internet environments, requiring only 256K of buffer memory.

To set proper redundancy levels and buffer sizes, we measured Internet UDP performance within the U.S. over a range of times and endpoints, and observed the following behavior:

Route	Latency: Min/Max/Avg	Peak Loss
In-State (MD)	~ 10 / 220 / 25 ms	~ 1%
Transnational (US)	~ 270 / 2600 / 350 ms	~ 4%

Rather than choose an arbitrary maximum-load scenario such as has been done with MIDI over an Ethernet LAN (Foss and Mosala 1996), we have instead used transmission of an actual piece of music for our “worst-case” evaluation.

1.14 MIDI over the Internet Performance Evaluation

Using our loss and latency measurements as a guideline, we calculated the appropriate theoretical parameters for transmitting a sample work, Liszt's Piano Sonata in B Minor, a 25-minute piece with approximately 40,000 MIDI events, including note-ons, note-offs, and pedal controls. Achieving a 1/8000 probability of a dropped event, or one every five minutes, implied 80 duplicates for the in-state connection, and 320 duplicates for the national connection. A buffer equal to or slightly greater than the maximum latency should have produced an uninterrupted performance, even during periods of peak musical density.

Though it was ultimately not possible to test this system in the context of a real long-distance master class, a similar test environment was constructed to determine if these theoretical solutions resulted in superior reproduction of remote piano performance in practice. We transmitted the entire Liszt sample work from a computer at the Peabody Conservatory in Baltimore to a receiving computer at the University of California, Berkeley, recording the results at the destination for comparative purposes. After modifying the demonstration system to address firewall behavior in the test network environment, the outcome proved quite successful. We found that substantially less redundancy was necessary than hypothesized, achieving flawless trans-continental playback with only 5 duplicates. Conversely, buffer sizes of between double and ten times the maximum latency proved necessary to ensure continuous playback. Time constraints prevented investigating the underlying causes for why practical operating

parameters differed so much from theoretical estimates. However, upon execution using these revised trial values, the fidelity of the system proved satisfactory. In fact, an event-by-event comparison of transmitted data to received data revealed an exact reproduction of the source.

1.15 Conclusion

Based on this analysis of the results, as well as other informal listening evaluations by trained musicians, it was concluded that transmitting live performance between acoustic pianos using MIDI over the Internet is indeed both feasible and audibly superior to other existing means of reproducing remote piano performance. However, for use in a real-time setting such as a Master Class, the responsiveness of the system would likely need to be improved, to a latency of no more than several seconds, in order to facilitate reasonably natural interaction between parties at different locations.

1.16 References

Beatnik. 1 March 1999. Beatnik Inc.—enhanced audio solutions. <http://www.beatnik.com>.

Bolot, J.-C. 1993. End-to-end packet delay and loss behavior in the Internet. *Computer Communication Review* 23 (4): 289–98.

Casey, M.A. and P. Smaragdis. 1996. Netsound. In *Proceedings of the International Computer Music Conference*, 143. Hong Kong: ICMA.

Chadabe, J. 1997. *Electric sound: The past and promise of electronic music*. New Jersey: Prentice-Hall.

Clark, D. and W. Lehr. 1999. Provisioning for bursty Internet traffic: Implications for industry and Internet structure. In *Proceedings of the MIT Workshop on Internet Quality of Service*. Boston, MA: Advanced Network Architecture Group, Massachusetts Institute of Technology Laboratory for Computer Science.

Comer, D. 1995. *Internetworking with TCP/IP*. 3d ed. Vol. 1, *Principles, protocols, and architecture*. New Jersey: Prentice-Hall.

Eliens, A., M. van Welie, J. van Ossenbruggen, and B. Schonhage. 1997. Jamming (on) the Web. *Computer Networks and ISDN Systems* 29 (8–13): 897–903.

Fober, D. 1994. Real-time MIDI data flow on Ethernet and the software architecture of MIDIShare. In *Proceedings of the International Computer Music Conference*, 447–50. Aarhus, Denmark: ICMA.

Foss, R., and T. Mosala. 1996. Routing MIDI messages over Ethernet. *Journal of the Audio Engineering Society* 44 (5): 406–15.

Loy, C. 1985. Musicians make a standard: The MIDI phenomenon. *Computer Music Journal* 9 (4): 8–26.

MIDI Manufacturers Association (MMA). 1996. *The complete detailed MIDI 1.0 specification*.

Miller, J. 1996. Simply grand. *Electronic Musician* 12 (11): 92–100.

Moore, F. R. 1988. The dysfunctions of MIDI. *Computer Music Journal* 12 (1): 19–28.

Ramanathan, P., and K. G. Shin. 1992. Delivery of time-critical messages using a multiple copy approach. *ACM Transactions on Computer Systems* 10 (2): 144–66.

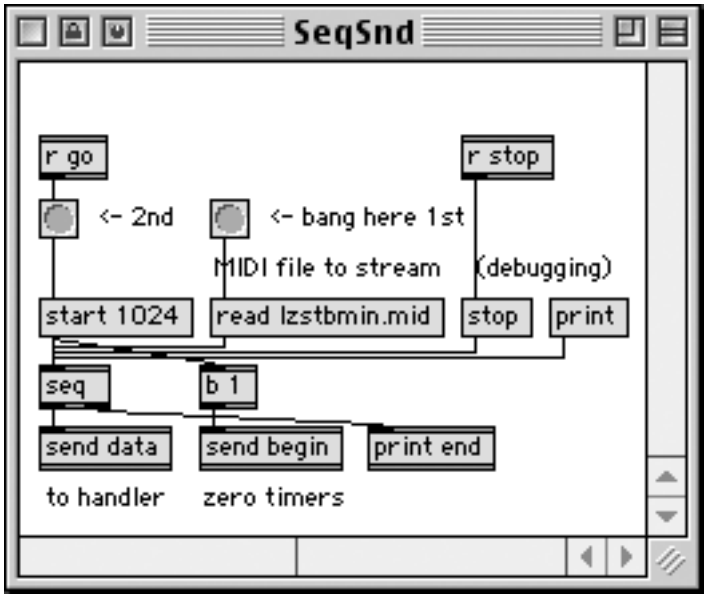
Scheirer, E. 1998. SAOL: The MPEG-4 structured audio orchestra language. In *Proceedings of the International Computer Music Conference*, 423–28. Ann Arbor, MI: ICMA.

Sun Microsystems. 1 March 1999. The source for Java technology. <http://java.sun.com>.

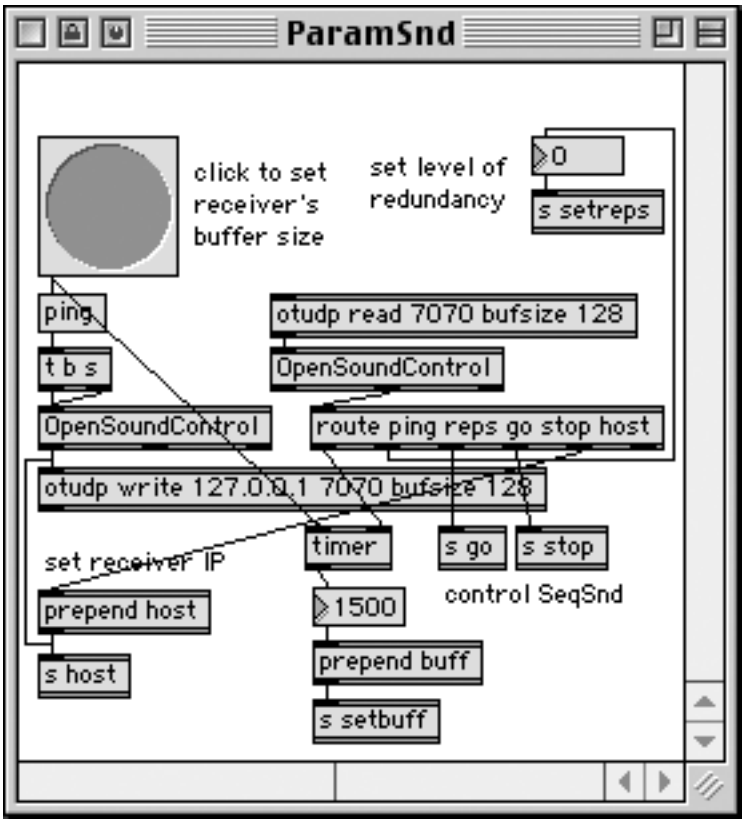
Wright, M. 1998. Implementation and performance issues with OpenSound Control. In *Proceedings of the International Computer Music Conference*, 224–27. Ann Arbor, MI: ICMA.

Zicarelli, D. 1998. An extensible real-time signal processing environment for MAX. In *Proceedings of the International Computer Music Conference*, 463–66. Ann Arbor, MI: ICMA.

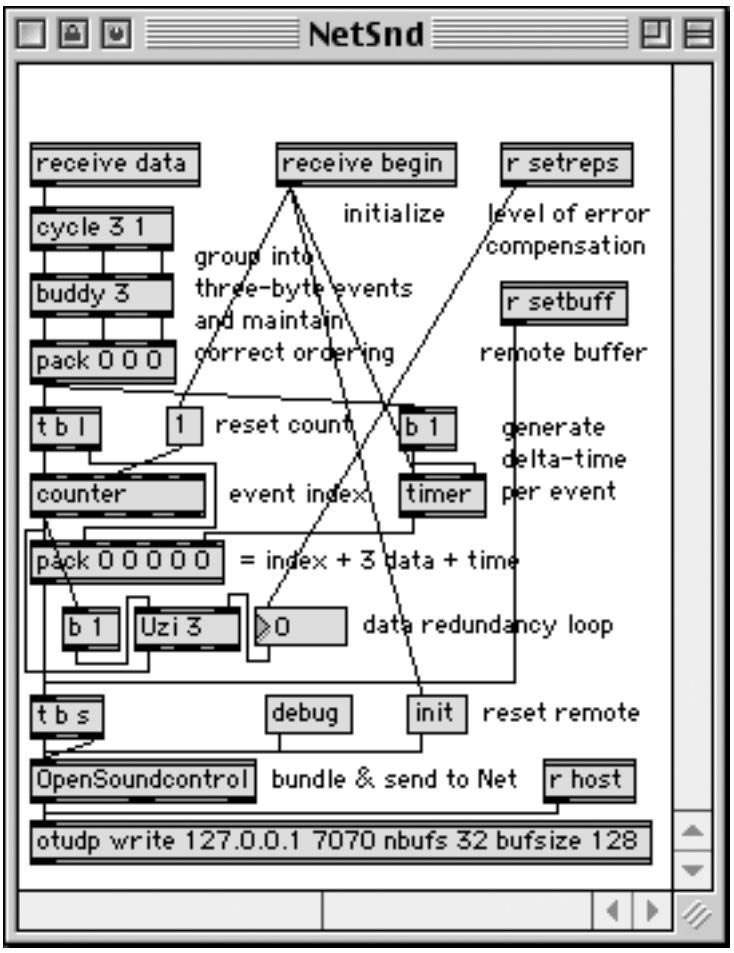
1.17 Appendix: MAX Patches



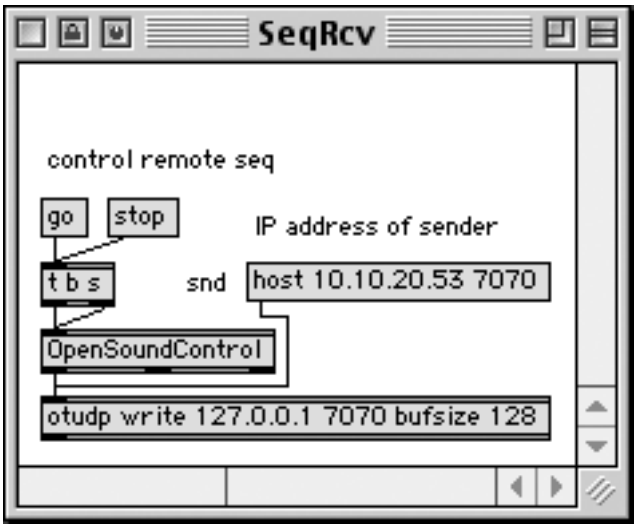
Sender: Triggering playback of a prerecorded MIDI sequence



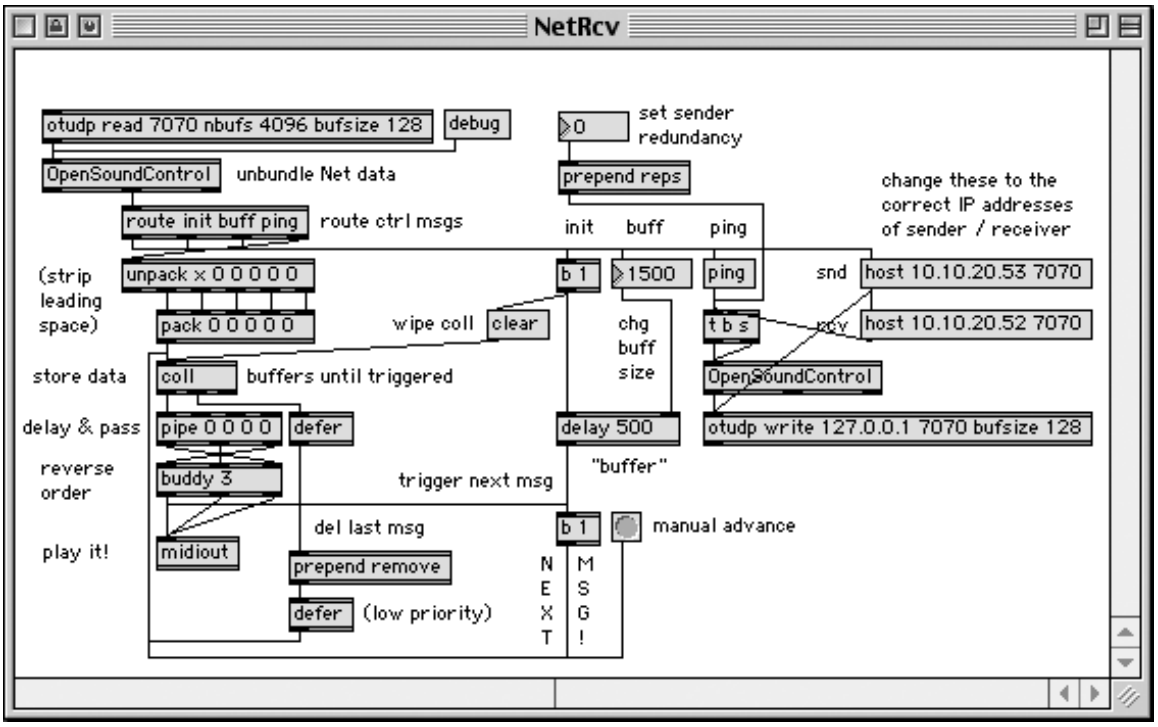
Sender: Testing the network connection and optimizing parameters



Sender: Adding time tags and redundancy to incoming MIDI events



Receiver: Controlling the sender activity remotely



Receiver: Collecting remote MIDI events into an indexed database and buffering output

Using the Web for Live Interactive Music

Cyberspace. A consensual hallucination experienced daily
by billions of legitimate operators, in every nation,
by children being taught mathematical concepts…
A graphic representation of data abstracted from
the banks of every computer in the human system.
Unthinkable complexity.
Lines of light ranged in the nonspace of the mind,
clusters and constellations of data.
Like city lights, receding….

— William Gibson, *Neuromancer*

2.1 Introduction

Telemusic #1 originated from a long-held artistic vision of Randall Packer, a composer, media artist, and Director of the Center for New Media at the Maryland Institute, College of Art (Packer 2000). Through the process of close collaboration and alignment of his compositional goals with my assessments of technical feasibility, it became realized as an interactive musical work incorporating live performers, signal processing, and real-time participation via a public Web site. Making this concept a reality required the extension and integration of existing technologies in ways that had not previously been documented. One primary objective for the software development was to adhere to open, established standards as much as possible. This mandate led to the use of Java and JavaScript (standardized as ECMAScript) for the core components, and the OpenSoundControl (OSC) objects (Wright 2001) from the UC Berkeley Center for New Music and Audio Technologies (CNMAT) for interfacing with Cycling '74 MAX (Cycling '74 2001). The resulting system enables anyone interacting with a Web site to send numeric data over the Internet to a computer running MAX, which can then process the input as it would from any other source. In a sense, one can thus transform the Web into a musical instrument, performed by anonymous visitors—an unseen ensemble of limitless proportions.

2.2 Background

Previous attempts to interface standard Web technologies with interactive music have involved substantial limitations. *Variations for WWW* (Yamagishi and Setoh 1998), the first successful integration of a Web site with MAX, was a breakthrough for its time. However, it only worked with form-submitted input, substantially limiting the flexibility and reactivity of the user interface, and was based on David Zicarelli's W protocol and W server, which are no longer supported. JSyn (Burk 1998), a plug-in expanding the audio capabilities of Java applets, is an excellent tool for producing interactive sound entirely within a local browser environment. This software was extended with TransJam (Burk 2000), a server-based module enabling communication between multiple JSyn users. However, using the system requires installation of the JSyn plug-in on each client, the TransJam server component is not available to the public, both are closed-source projects, and they do not include any facility for communicating with other common software applications such as MAX. *Piano Master Classes via the Internet* (Young and Fujinaga 1999) introduced musically sophisticated error-correction for near-real-time collaboration over the Internet. However, the musical elements of the system were entirely based in MAX, and could not be integrated with a browser-based interface.

The research supporting *Telemusic #1* aimed to pick up where these predecessors left off, enabling a Web-based interactive interface to control the musical capabilities of MAX/MSP in either a client/server or peer-to-peer fashion, using sophisticated error correction for transmitting live control data over the Internet. In addition, full access to

the relevant source code will be made available under the GNU General Public License (GPL), allowing freedom to manipulate the system so long as any modifications or enhancements are submitted back to the community under the GPL, for all to share (FSF 2001).

2.3 Inception

Telemusic #1, this collaborative intermedia work by Randall Packer with Steve Bradley and myself, was the driving force behind development of a distributed input environment functioning in parallel with a live performance. The physical aspects of the piece took place on Friday, November 3rd, 2000 at the Sonic Circuits VIII International Festival of Electronic Music and Art in St. Paul, MN, USA. A Web site, www.telemusic.org, was constructed for the event, presenting an interface for visitors worldwide to participate in real time. The primary controls were implemented in Macromedia Flash. Visitors to the site could define a 'telematic identity' and navigate through a virtual space. By manipulating various elements, they projected their actions into the physical space in the form of triggered samples, filter parameters, and other live sonic results, according to Packer's compositional framework for interpreting these control inputs. In addition, an active audio feed was streamed back to Web participants using RealAudio, providing a stereo rendering of the performance in progress. Thus it was possible to interact with the Web site and hear concrete audible feedback of one's actions in a matter of seconds. The Walker Art Center, one of the sponsors of *Telemusic #1*, also cooperated by covertly re-coding their home page, mapping embedded triggers to various links so that visitors to

their Web site became unwitting participants in the piece. The resulting work was a dynamic balance of precomposed elements and improvisation, as the input received from the Internet was inherently unpredictable. Based on our objectives for the performance and feedback from Web participants, it was a successful premiere, both technically and artistically.

2.4 Vision

The eventual goal of this research will be to dissolve boundaries between real and virtual space, by creating a persistent, distributed interactive world of sound and imagery. As development advances, it will naturally intersect to an increasing degree with efforts in the pursuit of 'virtual reality'. A survey of the literature in this field reveals a strong predilection towards representationalism—modeling the physical properties of our tangible world as realistically as possible. Much effort has been devoted to generating reasonable simulacra of the known world, in the expectation that immersion can best be achieved by recreating the familiar, with less interest demonstrated in constructing a compelling environment based on alternative principles. This may reflect a 'scientific' emphasis on analyzing and reproducing external features for judgement by comparison, as opposed to 'artistic' considerations of reinterpretation and internal consistency. Rather than focusing so much on anthropomorphic facsimiles of ourselves, recognizable visual environments, and spatially flawless acoustic cues, perhaps more investigation of compositional issues is warranted. Surely there are other ways of delivering a compelling experience to a participant other than emulating his/her natural milieu.

Most graphical virtual worlds to date have been designed as isolated entities, to be experienced by one or a few physically collocated individuals. Groundbreaking work in this area which laid the foundation for extending these experiences to a shared context should be acknowledged. *Sculpting 3D Worlds with Music* (Greuel et al. 1996) and related efforts by Fakespace Music on rendering real-time visual analogues to sound informed the parallel development of the dynamic audio-driven visualizations one can enjoy in any recent music-playing software. These visualizations can be extraordinarily compelling in the collective experience of a dance club—a virtual equivalent seems inevitable. *Coney Island* (Bargar et al. 2000) integrated virtual scenery with responsiveness to audience participation. During a performance of the work, the audience saw simulated first-person travel through a 'theme park' projected on a stage screen and could influence this environment by tapping on any of several drum pads in the house. Even though the work was confined to a single contiguous physical space, it nevertheless was a step towards assimilating distributed input into a unified collective experience.

Taking a leap from this elite demonstration of technology in the direction of more populist counterparts might lead us to the current crop of PC-based Massively Multiplayer Online Role-Playing Games (MMORPGs). These environments also enable participants to shape a shared world, but their audience is connected only by a network of client computers. MMORPGs are the culmination of well-established communal constructs born on the Internet in the form of Multi-User Dungeons (MUDs), and their related descendants. Originally text-based, the trajectory of these shared realities has been

towards ever-greater explicit visual detail, to the extent of even scanning one's likeness into the game, molding an avatar as an unmistakable extension of oneself. There is every reason to believe that these trends will continue, defining the state-of-the-art in consensual virtual reality. These game environments, though still aspiring to traditional, albeit fantastic, notions of realism, encompass many useful paradigms for less ambitious endeavors:

- Bidirectional communication between each interactor and the virtual space
- Independent and persistent existence of the virtual realm
- Consistent, perceptible rules governing interaction and feedback
- Aspects of emergent behavior—reaction of the virtual world incorporates, but is not limited to, direct manipulation
- Potential for coordinated collaboration with other interactors without requiring external channels of communication
- Evolution—the shared environment changes as a result of the sum total of interactions

These conditions describe many aspects of our perception of physical reality, but need not be implemented as a literal reflection thereof, with all the complexity that would imply. Taking as a point of departure the observation that music is deeply meaningful though fundamentally abstract, the features above can potentially be incorporated into a virtual environment arising from the same conceptual basis as our relationship to music.

2.5 Related Projects

Others have made forays into the territory of musically collaborative virtual space. Enterprises such as Rocket Network (Rocket Network 2001) enable participants to use central servers as a convenient conduit for sharing their material out of real time. Such services underscore the impossibility of simultaneous 'jamming' over the network in the traditional sense, without confronting the potential for manipulating audio in a time-insensitive context. *Cathedral* (De Ritis 1999) combines scheduled live performances with Web-based interaction, but the experience is unidirectional—each participant inhabits their own isolated sonic sphere, with no ability to influence the performance as witnessed by others. Interestingly, the Virtual Worlds Group at Microsoft Research may have progressed the farthest in this direction with their prototype, MusicWorld:

> … a graphical collaborative musical performance environment where avatars join together to explore a live, on-line 'album' of multiple songs. Within each soundscape, avatars can mix sounds and compose their own musical patterns that are shared in real-time. Even though all of the avatars can affect the music individually, all the changes they make are instantly updated across the network, guaranteeing that the music is truly collaborative and heard the same way for all participants. (Microsoft Research 2001)

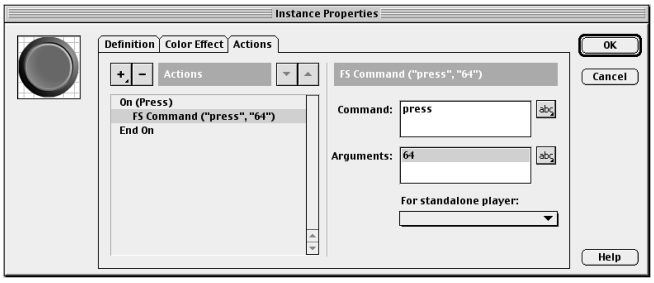
MusicWorld exhibits some important characteristics of the 'ideal' distributed musical environment described above, but is limited by its emphasis on literal representations and incompatibility with non-Windows platforms.

All of these efforts illustrate worthwhile objectives, but none seem to aspire to the larger goal of creating a persistent virtual space where anyone can collaboratively interact within a mutually composed and experienced environment for exploring the interface

between sound and image. There is no inherent obstacle to achieving such a goal. This paper represents a first step on that path.

2.6 Architecture

The original software implementation for *Telemusic #1* was narrowly focused on the task of enabling a Flash animation within any Web browser on the Internet to send data to a specific computer running MAX, with minimal delay. Addressing this initial requirement began with the Flash `FSCommand()`, a built-in method for addressing JavaScript that allows custom functions to be executed based on seven possible object behaviors in a Flash movie: Press, Release, Rollover, Rollout, DragOver, DragOut, and KeyPress. Here is a Flash inspector window attaching an `FSCommand()` to an object:



`FSCommand()` specifies two pieces of data—a command, and arguments. In this case the command is "press," signifying the action performed, and the argument is "64", the number that will eventually find its way to MAX.

When an `FSCommand()` is triggered in a Flash movie, the command and arguments are passed via the Document Object Model (DOM) (Flanagan 1998) to the host browser to be

processed by an appropriate JavaScript function. Here is a simplified JavaScript function to handle the `FSCommand()` above:

```
<SCRIPT LANGUAGE=JavaScript>
function TeleButton_DoFSCommand(command, args) {
   if (command == "press") {
   document.TeleApplet.press(args);
   }
```

This JavaScript simply acts as a middleman to catch data coming from the Flash movie and pass it on to a Java Applet—`TeleApplet`—embedded in the same page. Known as LiveConnect, this communication between plug-ins, JavaScript, and Java Applets originated with Netscape Navigator 3.x, and is now supported in most browsers. The `TeleApplet` also performs as an intermediary, except instead of making a local connection, it transmits long-distance, over the network. Here is the simplified Java code:

```
public class TeleApplet extends Applet {
   public void press(String args) {
      sendData(args);
   }
```

Details of the `sendData` method are not shown because it is simply a standard implementation of TCP/IP socket communication. At the other end of this link is a Java Application—`routeOSC`—listening on the host Web server. `routeOSC` receives the data from the remote client, reformats it into the OpenSoundControl protocol, and finally transmits it to MAX. Here is the basic idea, in pseudocode:

```
public class routeOSC {
  try {
    ss = new ServerSocket(port); // to receive TCP
    buf = new StringBuffer();
    String args = “”;
    ia = new InetAddress(host);  // destination IP
    ds = new DatagramSocket(ia); // to send UDP
    socket = ss.accept();        // listening...
    buf = new StringBuffer(socket.read());
    args = String.valueOf(buf);  // parse data
    toOSC(ds, args); // reformat and transmit args
  }
```

Again, details of `toOSC` are not shown because it mostly involves string handling and a standard implementation of UDP datagram communication. Of course, once the data arrives in MAX, it can be freely interpreted as control information no different from that arriving via MIDI, internal processes, the local graphical user interface (GUI), or elsewhere.

Hopefully this description has made it clear that the whole process from beginning to end is rather straightforward. Every attempt has been made to maintain simplicity and transparency from concept to implementation, so that others will be able to readily understand the existing code and adapt it to whatever new purposes they can imagine.

The data path may seem rather convoluted, but is necessitated by limitations of third-party software and the intentional containment of browser capability for security reasons. Flash does not include any means of external communication other than passing an `FSCommand()` to JavaScript. JavaScript has no networking functions of its own, so must call methods from a Java Applet in order to communicate with anything outside of the host browser window. Java Applets are constrained by the well-known security 'sandbox', which prohibits establishing a network connection with any machine but the one from which the Applet was loaded—the host Web server. Once the data is received by the Java 'listener' application running on the Web server, it is finally free to be reformatted for any supported protocol and sent to any machine on the Internet. The only way to circumvent all these steps would be to create a browser plug-in specifically

designed to violate the well-established browser security model in return for a modest increase in expediency.

After successful initial testing, more features were soon added to offer an increased range of interactive possibilities. Independent of Flash actions, data could also be sent in response to common browser event triggers including page loading (`onLoad`), clicking links (`onClick`), element rollovers (`onMouseOver`), etc. Also, in order to uniquely identify Web visitors, a means of writing cookies, and including that ID in each bundle sent to MAX, was developed. In theory, any interface that can be coded in JavaScript, as a Java Applet, or as a plug-in that can address the DOM, could successfully integrate into this system. Once the compositional challenges of inherent network latency and decoupled feedback to the user are accepted, the potential for realizing distributed interaction is boundless.

2.7 Limitations

One caveat with the software is that it is currently limited to using the OSC protocol developed at UC Berkeley CNMAT to communicate with MAX. Despite its moniker, OpenSoundControl is not entirely open. Much of the general specification has been published (Wright 1998), but source code to the specific implementation used by the MAX external objects is not available, and substantial portions of the software fall under copyright of the UC Regents, complicating use of the code in other projects. However, OSC, and OTUDP, its companion object for UDP-based network communication, are the

only networking externals currently supported for the MAX environment, and they function quite well in this capacity. OSC/OTUDP will continue to be the primary target, but interoperability will be extended to the open-source projects Pd by Miller Puckette (Puckette 1997) and jMax from IRCAM (Dechelle et al. 1999) as well, for those working in these environments where complete end-to-end control of software is desired.

This system was designed for maximum cross-platform compatibility. It has been successfully tested using Netscape Navigator under Windows, MacOS, and Linux, and Microsoft Internet Explorer (IE) under Windows. IE under MacOS does not work because current versions of IE:Mac unfortunately do not properly support LiveConnect, thus communication between JavaScript and Java is broken. The server application has been successfully tested under Windows, Linux, and MacOS X. Because most of the software is written in Java, it is quite portable, and every effort will be made to ensure ongoing compatibility with common operating systems and browsers.

2.8 Ongoing Development

My collaboration with Randall Packer continued in the form of his next intermedia work, *Telemusic #2*. This piece builds technologically on its predecessor by interpreting network traffic statistics gathered in real time to generate an organically textured musical atmosphere as a canvas for interaction. A prototype installation, with additional MAX programming by Margaret Schedel, was demonstrated at the “Music Without Walls?

Music Without Instruments?" conference at De Montfort University, Leicester, UK from June 21–23, 2001.

The software required was inspired in part by two open-source projects: the Multi Router Traffic Grapher (Oetiker 2001), a tool for aggregating Simple Network Management Protocol (SNMP) queries, and Peep (Gilfix and Couch 2000), a framework for auralizing dynamic network conditions. While there was hope of sampling aggregate Internet traffic from a nearby Internet Service Provider (ISP), the procedural obstacles proved insurmountable. As a result, the prototype monitored network usage at NOVA Research Company, in Bethesda, MD, USA, a small business where I have full authority over the infrastructure, and substantial network traffic is generated during the workday. Raw statistical data was continuously collected using TCPSTAT (Herman 2001) on a GNU/Linux platform. The bursty nature of network activity was smoothed out by sampling over a two-second window, outputting metrics such as total number of bytes, total number of packets, and breakdowns by protocol such as ARP, ICMP, TCP, and UDP. A Java parser application digested the data, formatted it into OSC, and sent it across the Atlantic Ocean to MAX. More extensions to the `TeleApplet` were also written, to enable transmission of a Web client's IP address and a time stamp in addition to the Flash and browser triggers used previously.

This prototype of *Telemusic #2* was received very positively, with the ebb and flow of network utilization providing an intuitive audible representation of the passage of time at the source, particularly in the contrast between day and night. Looking beyond the

horizon, Packer sees *Telemusic #3* adding musically responsive visualizations to the environment to create a more immersive intermedia experience.

2.9 Future Directions

An initial alpha release of Web2OSC has already occurred. The primary milestones for the first public beta version are to continue adapting the code from its orientation of fulfilling specific requirements towards supporting more general-purpose functionality, and to provide adequate documentation.

Beyond providing a stable and comprehensible code base with which others can experiment, there are numerous other features that have been envisioned:

- Data flow is presently one-way, from browser to MAX only. It would not be difficult to send a response back from MAX to the client Java Applet, allowing bidirectional communication and greater interactive potential.
- It would be prudent to add handshaking and/or error-correction between the `TeleApplet` and `routeOSC`, for the sake of both data integrity and security.
- `routeOSC` is currently configured using command-line options. A GUI for setting parameters and making dynamic adjustments would be helpful.
- It is possible to re-broadcast incoming data in MAX, but it would also be useful to enable `routeOSC` to send to multiple destinations.

Ultimately, I would like to see Web2OSC become more than just a collection of code. As I pursue my own efforts to compose Internet-based works, I expect to build upon this humble beginning towards a tool worthy of being called a distributed virtual instrument.

2.10 Conclusion

The system described herein has been designed for flexibility and extensibility, offering a framework that can support many potential applications in addition to those already noted. I sincerely hope that development and sharing of this software can inspire further creative use of the Web as a means of participating in live interactive works. Feedback and feature requests are always welcome. The latest version and documentation can be found at <http://www.netmuse.org>.

2.11 References

Burk, P. 1998. JSyn—a real-time synthesis API for Java. In *Proceedings of the International Computer Music Conference,* 252–55. Ann Arbor, MI: ICMA.

———. 2000. Jammin' on the web—a new client/server architecture for multi-user musical performance. In *Proceedings of the International Computer Music Conference*, 117–20. Berlin, Germany: ICMA.

Bargar, R., F. Dechelle, I. Choi, A. Betts, C. Goudeseune, N. Schnell, and O. Warusfel. 2000. *Coney Island*: Combining jMax, Spat and VSS for acoustic integration of spatial and temporal models in a virtual reality installation. In *Proceedings of the International Computer Music Conference*, 149–53. Berlin, Germany: ICMA.

Cycling '74. MAX/MSP. 10 December 2001. <http://www.cycling74.com/products/maxmsp.html>.

Dechelle, F., M. De Cecco, E. Maggi, and N. Schnell. 1999. jMax recent developments. In *Proceedings of the International Computer Music Conference,* 445–48. Beijing, China: ICMA.

De Ritis, A. 1999. *Cathedral*: an interactive work for the Web. In *Proceedings of the International Computer Music Conference*, 224–27. Beijing, China: ICMA.

Flanagan, D. 1998. *JavaScript: The definitive guide*. 3d ed. Sebastopol, CA: O'Reilly & Associates, Inc.

Free Software Foundation (FSF). 15 July 2001. GNU General Public License. <http://www.gnu.org/copyleft/gpl.html>.

Gilfix, M., and A. Couch. 2000. Peep (the network auralizer): Monitoring your network with sound. In *Proceedings of the 14th Systems Administration Conference*. New Orleans, LA: USENIX.

Greuel, C., M. Bolas, N. Bolas, and J. McDowall. 1996. Sculpting 3D worlds with music; advanced texturing techniques. In *Proceedings of the SPIE—The International Society for Optical Engineering*. 2653: 306–15. San Jose, CA: SPIE.

Herman, P. 1 May 2001. TCPSTAT. <http://www.frenchfries.net/paul/tcpstat/index.html>.

Microsoft Research. 1 May 2001. MusicWorld. <http://www.vworlds.org/music/>.

Oetiker, T. 1 May 2001. MRTG—the multi router traffic grapher. <http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/paper/>.

Packer, R. 2000. 1 May 2001. Live space for collective electronic music performance. <http://www.telemusic.org/Telemusic_1.html>.

Puckette, M. 1997. Pure data. In *Proceedings of the International Computer Music Conference,* 224–27. Thessaloniki, Greece: ICMA.

Rocket Network. 10 December 2001. Rocket Network. <http://www.rocketnetwork.com>.

Wright, M. 1998. Implementation and performance issues with OpenSound Control. *Proceedings of the International Computer Music Conference,* 224–27. Ann Arbor, MI: ICMA.

———. 10 December 2001. OpenSoundControl. <http://cnmat.cnmat.berkeley.edu/OSC/>.

Yamagishi, S., and K. Setoh. 1998. *Variations for WWW*: network music by MAX and the WWW. In *Proceedings of the International Computer Music Conference,* 510–13. Ann Arbor, MI: ICMA.

Young, J., and I. Fujinaga. 1999. Piano master classes via the Internet. In *Proceedings of the International Computer Music Conference,* 135–37. Beijing, China: ICMA.

Application of Fault Tolerance to Interactive Music

Things fall apart; the centre cannot hold…

— W. B. Yeats

3.1 Introduction

Ever since humans first fashioned tools, we have had to consider the reliability of those tools and cope with the consequences of their failure. The unprecedented complexity of our tools in the digital age has given rise to the study and practice of fault tolerance (FT), with the objective of delivering acceptable operation even during sub-optimal circumstances. Over the past fifty years, FT has steadily advanced in stride with the permeation of computers into all aspects of society and human welfare. As we become more deeply enmeshed in the web of our own machinery, the repercussions of its failure grow more profound.

The study of FT as we know it today emerged from the science of information theory. In the course of a decade, Claude Shannon (Shannon 1948), Edward F. Moore (Moore and Shannon 1956), Richard Hamming (Hamming 1950), and John von Neumann (von Neumann 1956) developed the principles of error-correction and redundancy. These principles were immediately put into practice by the fledgling telecommunications, computing, and avionics industries in pursuit of reliability (Siewiorek and Swarz 1998). William H. Pierce unified theories of masking redundancy (Pierce 1965), and shortly thereafter Algirdas Avizienis integrated these techniques for detection, diagnosis, and recovery into the concept of fault tolerant systems (Avizienis 1967). In 1971, the NASA Jet Propulsion Lab (JPL) and the Institute of Electrical and Electronic Engineers (IEEE) sponsored the first International Symposium on Fault-Tolerant Computing, engendering the IEEE Technical Committee on Fault Tolerant Computing (TCFTC), which continues

to be a major forum for discourse and progress (Avizienis 1997). From 1985 to 1992, Jean-Claude Laprie led a team of members from the International Federation for Information Processing (IFIP) Working Group 10.4 to publish the definitive reference work in the field (Laprie 1992). FT has since been most prominently employed in high-profile, high-cost areas such as telecommunications (Levendel 1994), defense (DARPA 2003), aeronautics (Torres-Pomales 2000), space exploration (Hecht, Hecht, and Shokri 2000), nuclear power (Hecht et al. 1991), and financial services (Birman 1993). However, FT is increasingly relevant at all levels of computing, wherever the price of system failure is greater than that of prevention.

Electronic music has likewise evolved in tandem with the increasing power and ubiquity of computing hardware. At first, computers were used to generate sound offline, as a digital equivalent of tape, and in that context their dependability was not a critical issue (Chadabe 1997, 113). However, since machines have become capable of participating in live interactive electronic music (IEM), they have been expected to respond appropriately in real-time, just as a traditional instrument or ensemble partner would (Madden et al. 2001). In this environment, a system crash during performance is transformed into the rough equivalent of someone dying onstage. Such an occurrence—*musicus interruptus*—is unpleasant for everyone involved. The composer is disappointed because her musical idea has been undermined. The performer is anxious because his instrument has vanished and he is all but powerless to recover gracefully. The audience is frustrated because they would have rather spent their time and money on art or entertainment that doesn't fail, and at least gives them the opportunity to evaluate its creator's full intent. This paper

aims to help prevent these negative experiences, and to empower computer musicians with the idea of tools that they can trust, engendering an environment in which composers feel liberated to explore their creative vision, and performers have confidence to play their instruments without reservation.

While few would argue that IEM represents the same potential for catastrophe as a manned space mission (NASA 1999) or air-traffic control system (Cristian, Dancey, and Dehn 1996), the presentation of an interactive piece can have a similar quality of breathtaking precision, in which everything must function perfectly to achieve success. Even on a budget somewhat below the level of key public infrastructure, the concepts and techniques developed in pursuit of FT can be usefully applied to the compositional design and performance of an interactive musical work. The nature of sound leaves little margin for error in maintaining the suspension of reality which a flawless performance can convey (Truax 2001, 15). By incorporating practical elements of FT into the entire creative process, the experience of IEM can be improved for composers, performers, and audiences alike, bringing the quest for that elusive perfect realization closer to fruition.

3.2 Principles of Dependability

As with any maturing discipline, an extensive vocabulary of FT has evolved in parallel with the codification of fundamental principles and the emergence of new strategies that build upon them. To facilitate the explanation of how FT can be applied to IEM, I will first present a primer on widely accepted FT terms and concepts.

3.2.1 Fault Definition

Intuitively, one would think that a fault is simply a bug in either hardware or software, but it is not quite so straightforward. One way of viewing the definition follows the consequences backwards—a **failure** occurs when the system delivers service that deviates from the specified functionality, generally expressed in a way that is apparent to a user of the system. A failure can range from a single incorrect value to a complete system halt. A **fault** is the ultimate cause of such a failure. A third term, **error**, is often used, with some ambiguity, to refer to the intermediate state where a fault has been exposed but has not yet resulted in a failure (Laprie 1995). For example, a defective memory cell is a **dormant fault** if a value stored there by a program will be corrupted. Once a value is written, the corruption does not become apparent until the value is read back from that cell, at which point the fault is **triggered**, becoming an **active fault**, and results in an error—an incorrect value returned by the operation. If the error is corrected via either hardware or software mechanisms, the program proceeds normally and the memory fault becomes dormant again (Heimerdinger and Weinstock 1992). If the error leads to incorrect behavior, then failure occurs. It may take a cascade of many faults to produce an error, and the accumulation of numerous errors before a system fails. These periods between fault activation and failure manifestation define the window of opportunity for recovery, where FT comes into play.

Faults take many forms, defined both by their origin and their expression. A **hardware fault** originates in the electromechanical architecture of a system, such as the silicon circuitry of a CPU or memory chip, the spinning platters of a hard disk, or the armature of a robotic servomotor. A **software fault** originates in the instructions that are executed by the hardware, such as an operating system, application program, or communications protocol (Laprie 1995). A **persistent fault** occurs consistently under the same conditions and can often be diagnosed via testing procedures, while a **temporary fault** is transient and difficult to reproduce. These types of faults can be further described by their circumstances. An **environmental fault** occurs because of stress on the system such as excess heat, electrical fluctuations, or spilled coffee. Related to this is a **specification fault**, in which the requirements did not properly anticipate the usage of the system, or the system is subjected to use outside the scope of the specifications. An **integration fault** is the product of interrelationships between system components, and is often the most complex and insidious type of fault to isolate and repair. An important but often overlooked threat is the **documentation fault**, in which the documented system does not accurately describe the real system. A key concern in IEM is the **timing fault**, in which the system misses a deadline, by delivering a service or result either too early or too late (Heimerdinger and Weinstock 1992). Numerous other fault categories have been put forth, but those above are relatively universal, and sufficient for discussing the problems encountered, and the remedies available to the audience of this paper.

In modern commercial-off-the-shelf (COTS) computers, software failures are far more prevalent than hardware failures. A brief analysis reveals the reasons. Hardware faults

tend to be persistent, so rigorous testing by the manufacturer enables identification and repair of most of them. Environmental requirements of COTS hardware—e.g. temperature, altitude, humidity, acceleration—are clearly defined and tend to be the same as those necessary for human comfort. Interactions amongst hardware tend to proceed using highly engineered, standardized, and documented interfaces, because few hardware products can survive in the marketplace without ensuring complete interoperability. By contrast, faults in COTS software are so common that they are routinely accepted as an inevitable feature of the computing landscape, but programmers do not deserve all of the blame. Non-trivial software faults tend to be temporary and difficult to diagnose. A software environment has vastly more variables than our physical one. The marketplace encourages the addition of new features in each revision rather than optimization of existing functions, and rarely punishes software interfaces that are poorly defined, non-standard, or frequently changing (Siewiorek and Swarz 1998). Finally, most mature software must maintain backwards compatibility with its previous iterations, often leading to a situation in which the newest, best code must rely on the oldest, worst code for its operations. Fortunately, there are options for coping with software faults in COTS systems to make these weaknesses manageable and achieve reasonable dependability.

3.2.2 Dependability Objectives

Jean-Claude Laprie, one of the primary architects of fault tolerance terminology, defines **dependability** as "that property of a computer system such that reliance can justifiably be placed on the service it delivers (Laprie 1995)." Just as there are numerous interrelated

but distinct types of faults, there are also subdivisions of this gestalt concept, objectives which may help focus emphasis when building or evaluating a system. **Reliability** relates to continuity of correct service, or conversely, to the time until failure, with the recognition that no system can be 100% reliable. **Availability** relates to the readiness of the system to deliver correct service, in the context of necessary usage. A business database that was only required to be online five days a week could demonstrate high availability even if it was routinely out-of-service for maintenance on weekends. **Safety** relates to the absence of catastrophic consequences on the user(s) and/or the environment. It may be counterintuitive, but a system can have high safety even with low reliability, if failure modes are properly contained and controlled. An average car, for example, inevitably fails in a wide variety of ways over the course of its lifetime, but rarely are failures of the automobile system itself responsible for the low safety we associate with car travel. **Integrity** relates to the absence of improper alterations to system state or information. Generally integrity is determined by the internal operation of the system. **Security** becomes a factor when an external system or user is involved, and the question of authorization arises, as well as **confidentiality**, relating to the disclosure of sensitive data. **Maintainability** relates to the ease of making repairs and/or modifications to the system, as well as the impact of such maintenance on other aspects of dependability (Laprie 1995). Finally, there is the specialized notion of **robustness**—dependability with respect to erroneous input—that is particularly applicable in the context of IEM (Avizienis, Laprie, and Randell 2001).

The integrative notion of dependability has also been called **trustworthiness**, which more clearly communicates both the quantitative and qualitative aspects of the concept (Avizienis, Laprie, and Randell 2001). System designers and operators probably think of dependability mostly in terms of metrics—error rates, safety record, mean time between failures (MTBF), data consistency, or overall uptime. These are obviously important statistics, but unfortunately they can only truly be measured and have relevance in retrospect. While a developer may be able to tell you that systems *similar to yours* have performed flawlessly, that doesn't ensure that your system will demonstrate the exact same behavior (Laprie 1995). So an equally critical expectation is predictability upon failure, which is often known as **graceful degradation** (Herlihy and Wing 1991). For example, a **fail-safe** system is designed to preserve safety in all possible scenarios. A nuclear reactor must be fail-safe against human error, malicious attack, natural disaster, and anything else we can think of. A **fail-op** system aims to provide a useful subset of its specified functionality despite failure of some components. The Internet can be considered fail-op because even if many key nodes fail, it should still be able to provide network connectivity by redirecting packets through other functioning routers, continuing to deliver service with reduced performance. A system designed to have **no single point of failure** will function normally in the event of any individual subsystem failure. Such systems are usually also designed to facilitate replacement or repair of the failed subsystem while the system remains operational, prior to occurrence of another failure. A **fail-silent** system will never produce incorrect values, opting instead to return no value at all if it cannot be verified correct. A surgical telepresence system is an environment in which no action is clearly preferable to a wrong action. A **fail-stop** system requires a

restart after any failure (Heimerdinger and Weinstock 1992). Real-time systems are rarely designed as fail-stop, but applications in which guaranteed perfect results are paramount, such as complex physics modeling or biological process simulation, might sensibly be implemented as such. These subjective goals demonstrate that dependability is not only about keeping systems running, but also knowing what will happen when a failure does inevitably occur.

3.2.3 Dependability Strategies

Strategies for achieving dependability goals fall into four broad categories: fault prevention, fault forecasting, fault removal, and fault tolerance. **Fault prevention** aims to avoid the occurrence or introduction of faults by employment of quality control methods during the system design phase. Such methods might include rigorous specification and requirements analysis, modularization, radiation shielding, or advance operator training, dependent on the deployment context and the potential faults of highest concern (Avizienis, Laprie, and Randell 2001). **Fault forecasting** attempts to predict the future incidence and probable consequences of faults in completed systems (Laprie 1995). The primary approach to fault forecasting is system testing and evaluation, via simulation of an operational environment, or intentional attempts to stress the system, known as **fault injection**. Fault injection can be applied to both hardware and software, either as an external disturbance at run-time, or by routines built-in during design for this purpose. Hardware fault injection generally includes artificially extreme environmental conditions such as high temperature or radiation intensity to determine an acceptable buffer zone

and likely failure modes, or manipulation of incoming data corresponding to a failed sensor or other input device. Software fault injection generally follows the same paradigm, by inducing extreme workload demands, timing out processes, triggering exception handlers, or manipulating input/output (Hsueh, Iyer, and Tsai 1997). **Fault removal** uses verification and validation to detect and correct faults when possible, both during development and in operation (Avizienis, Laprie, and Randell 2001). The TCP/IP protocol incorporates fault removal by verifying each incoming packet against its attached checksum—if the test fails, it requests a re-send of that packet, so that a fault in the physical layer (i.e. Ethernet) does not propagate to the operating system (Comer 1995, 191). Fault removal could also be as simple as noting a discrepancy between system behavior and the documentation, and modifying one or the other accordingly. Finally, **fault tolerance** aims to deliver correct service despite the presence of faults (Heimerdinger and Weinstock 1992). Because it is generally understood that in systems of any real complexity, no combination of techniques can ever be sufficient to eliminate all faults, dependability ultimately always becomes a question of fault tolerance.

3.2.4 Fault Tolerance

FT can be thought of as a bilateral approach—assessment and action—broken down into the following: detection, diagnosis, containment, masking, compensation, and repair. **Fault detection** is the process of determining that a fault has occurred. **Fault diagnosis** is the process of determining the 'actionable' cause of a fault, or deciding which subsystem is no longer dependable. **Fault containment** aims to stop the propagation of a fault from

its origin at the earliest point possible. **Fault masking** aims to maintain the integrity of a defined subsystem boundary by passing only correct data to other subsystems. In the event of a subsystem failure, **fault compensation** seeks to preserve functionality of the system as a whole using alternate resources. Finally, **fault repair** attempts to recover from a fault to the extent that the future FT capability of the system is not diminished (Heimerdinger and Weinstock 1992). In the absence of fault repair, an initially resilient system will deteriorate over time as a result of fault damage until it becomes increasingly brittle and prone to failure.

3.3 Dependability-Explicit Development Process

Now that the definitions, objectives, and general strategies of dependability have been enumerated, we can move on to discussion of specific manifestations of these ideas and their applicability to IEM. The conceptual approach naturally subdivides into the life-cycle phases of a system—specification, design, development, implementation, testing, and deployment—and the concrete solutions follow these stages as well. Creating a system when time and money are finite usually involves compromise, and often getting the core functionality "up and running" as quickly and inexpensively as possible becomes the prime objective of the project, at the expense of adequate dependability. Instead of treating fault awareness as a luxury, it is possible to integrate it into the construction process to facilitate the strategies described above. Such an approach is known as dependability-explicit development (Laprie 1995).

3.3.1 Specification

The first step in building a system is specification, defining the requirements of the system in terms of functions, resources, and dependability objectives. Specifications should incorporate knowledge of the immediate needs and the long-term evolution of the system, including portability and interoperability expectations. Non-essential functions should be minimized, as they increase the potential for faults, and are less likely to be thoroughly tested than core functions. End-users should be involved in the requirements definition as much as possible, in order to prevent human-interaction faults (Kaâniche, Laprie, and Blanquart 2000). In an artistic endeavor, such as composing IEM, it can be difficult or impossible to rigorously delineate the requirements of a goal that may be very nebulous at first, and change significantly between initial inspiration and ultimate realization. In fact, it is often essential to the artistic imperative to follow seemingly divergent threads and tangents, as the true essence of a work may become revealed as a product of the creative struggle itself rather than the solution to a preconceived problem. In this case, the specification phase becomes iterative with subsequent phases, until the prototype(s) converge towards a stable result. At that point, the composer shifts perspective from visionary to architect, and a useful set of specifications can be drafted and refined. There are at least as many compositional pathways as there are composers, but even the most chaotic approach to realizing a work can garner dependability benefits from a specifications document, perhaps by considering the solidification of an idea not as the conclusion, but as the beginning of a creative process.

3.3.2 Design

Once an acceptable specification has been conceived, the design phase can begin. An architecture is chosen that allows the system requirements to be fulfilled, including hardware and software. If specification is about winnowing infinite possibilities into concrete goals, design is about finding particular solutions for the problems raised by the specification. As the predominant number of system faults can be traced back to design flaws, this may be the most critical phase for dependability implications (Kaâniche, Laprie, and Blanquart 2000). Design can be broken into three major aspects: system structure, system behavior, and communication among components, each of which presents opportunities for fault prevention.

Building a dependable system structure relies on choosing dependable hardware and software, which ideally has been thoroughly tested by a reputable vendor who can provide quantitative data to demonstrate its reliability. Most commercial hardware fulfills these standards, but it is quite rare to find explicit dependability measurements of software outside of mission-critical fields. Thus it is necessary to rely on qualitative assessments—third-party reports, personal experience, and word-of-mouth. For specialized applications such as IEM, the software options can be so few that functionality must outweigh inherent dependability, which should be acknowledged so that other steps can be taken to offset the compromise.

Behavior—what must the system do?—is the core element of design, which generally drives all other decisions. Conceiving dependable behavior begins by recognizing the context in which the system will operate, but certain characteristics pertain whether the dominant interactions will be with humans or other machines. When we imagine idealized computers, we think of them as being logical, predictable, and consistent, and these qualities do reinforce dependability. Additional goals should be transparency, such that functions and results are not mysterious, descriptiveness, so that differences between expected and actual behavior are immediately apparent, and linearity, in which the magnitude of an action is reflected by the magnitude of reaction (Lanier 2002). In IEM, where systems may be designed to mimic aspects of human behavior, following such mechanistic principles might be contrary to a desire for autonomous or emergent properties. However, the responses of the system may be non-deterministic even while its operation is rigidly defined. For example, if a simple synthesizer module is programmed to randomly choose an arpeggio from a variety of patterns, it may respond differently to the same note played the same way in consecutive tests. However, it probably shouldn't begin singing "Bicycle Built for Two," just as this might be considered aberrant behavior from a human accompanist.

Communication between system components, also known as input/output (I/O), can either reinforce or undermine the dependability of structure and behavior. System I/O encompasses all internal and external interactions, including passing values between functions or objects, receiving feedback from sensor devices, and producing a response in any form. Every instance in which information is exchanged—human to hardware,

hardware to hardware, hardware to software, software to software, software to hardware, or hardware to human—presents the potential for information to be corrupted between sender and receiver, introducing a fault into the system. The system design should reflect the critical role of I/O by following standard interface and protocol definitions, and incorporating error-correction wherever possible. Many interactive music systems are designed to operate in real-time, such that a timing fault is more problematic than a value fault, and I/O priorities must be adjusted accordingly.

The full range of approaches to successful system design lies beyond the scope of this paper, but in the context of dependability the considerations can be thought of as practicing fault prevention, providing fault forecasting, preparing for fault removal and planning for fault tolerance. In other words, dependability must be a factor in every design decision. Faults should be preempted by meticulous choices wherever possible. The design should include comprehensive testing and status reporting capabilities. The system should be designed so that modifications to one component require minimal or no modifications elsewhere. Finally, the system design should incorporate appropriate FT routines at every level in order to meet the quantitative and qualitative FT objectives as explicitly defined in the specification.

3.3.3 Development

Paradoxically, designing FT into a system requires adding extra functions, that could potentially introduce additional faults and diminish the dependability of the system rather

than increasing it. One of the strategies for minimizing this risk is to develop the system using self-contained modules that can be verified independently, which in the realm of software suggests object-oriented programming (OOP). Most object-oriented languages even include a built-in measure for FT by way of exception handling. An exception is 'thrown' whenever an object receives input it wasn't designed to process, at which point control is passed to a 'handler' routine that can evaluate the fault and proceed accordingly. Exception handling can thus aid in fault detection, diagnosis, and containment, but this may not be sufficient in itself (Romanovsky, Xu, and Randell 1996). OOP also facilitates other common FT techniques involving redundancy and diversity. The most basic type of redundancy—multiple copies of the same object—can improve dependability, but there is considerable risk that the same fault might damage all the copies in the same way. A more sophisticated redundancy incorporates diversity, in which multiple versions of the same object address the same functionality through differing means (Xu et al. 1995). For example, an algorithm that exposes a CPU flaw by computing (a*b)*c would encounter the same fault every time, while an algorithm that executes a*(b*c) might circumvent the flaw. Such redundant objects can function in parallel, in which case any disagreement between them must be adjudicated to determine the faulty from the correct output. This approach is known as masking redundancy, because it attempts to hide faults by masking their presence with known valid results. Masking redundancy tends to be best suited to environments in which sufficient extra processor time and data storage are available to support the overhead of parallel execution. Redundant objects can also function consecutively, in which a faulty object shuts down while signaling to another object to take its place. This approach is known as

dynamic redundancy, and tends to be suited to real-time and distributed applications (Moser, Narasimhan, and Melliar-Smith 1996).

One of the methods for achieving redundancy and diversity during design and development is by following the open-source paradigm. Open-source software projects allow unlimited public access to their code, with the stipulation that any modifications to the code be made public as well (OSI 2003). An open-source community essentially leverages human redundancy and diversity in the service of those same qualities in software. A healthy community of developers working from different perspectives towards the same goal bears the potential for a level of effort that would be impossible via traditional means of financial compensation. One advantage of the open-source model can be increased fault removal through iteration and peer-review, famously stated as "Given enough eyeballs, all bugs are shallow." (Raymond 1999). Another advantage can be fault prevention through expertise and enthusiasm—community members tend to gravitate to projects that are interesting, rewarding, and provide a good opportunity to leverage their skills. The challenge of an open-source approach is forming and maintaining the requisite community to accrue these advantages. There are too many examples of active open-source IEM software projects to list here, but those that successfully compete with commercial products include Pd (Puckette 1997), jMax (Dechelle et al. 1999), RTcmix (Topper 1999), Csound (Vercoe 1990), and soon SuperCollider (McCartney 1998). Such projects with broad appeal tend to get the majority of attention, while esoteric applications can easily languish without ever reaching critical mass (Lawrie and Jones 2002). However, as an alternative to

traditionally regimented development, especially in the absence of substantial economic resources, the open-source model has the potential to significantly enhance dependability.

3.3.4 Implementation

In the implementation phase, prototypes transition from incubation to testbed, and independent objects and components begin to be integrated into a complete system. Assembling the pieces is an exercise in itself, as integration issues inevitably arise, requiring adjustment of interfaces and assumptions to achieve interoperability. At this stage, fault forecasting comes into play, as the real failure modes of the system are observed and organized in order to refine a fault tolerance plan. Basically, potential failures are categorized by severity and likelihood, so that the most common and problematic failures are slated to receive the highest FT priority (Kaâniche, Laprie, and Blanquart 2000). In an interactive music work, for example, clicks and aliasing may be the most frequent failures, with a statistical likelihood near 100% for the duration of the piece. However, if a 25% chance of sustained audio interruption is also noted, that may prove to be a more critical target of FT. Forecasting the range of failures from unacceptable to trivial not only focuses development resources, it also frames the subsequent testing methodology and evaluation analysis, ultimately yielding the standards by which the dependability of the project can be judged.

3.3.5 Testing

Testing ordinarily can cover the gamut from informal imitation of expected usage to rigorous use of automated tools that repeatedly invoke and monitor every function and interface. In the context of dependability, testing makes use of fault injection for the purpose of fault removal. Fault injection can take many forms. In the conceptual and design phases, simulation-based fault injection can assess theoretical strengths and weaknesses of a system in order to improve the design. Obviously the effectiveness of this approach depends on the accuracy of the simulation. Prototype-based fault injection takes place during development and implementation, by introducing faults either at the hardware level (logical or electrical faults) or the software level (code or data corruption). Another form of fault injection includes tools for artificially constraining memory available to an application, placing a competitive load on the CPU, and generating I/O traffic, common areas where diminishing resources can trigger faults. By stressing the system beyond normal expected conditions, this approach can reveal flaws or instabilities that might otherwise occur only sporadically (Dawson et al. 1996). Measurement-based analysis collects actual operational data from a sufficiently realistic test environment, enabling correlation of failure conditions to indicate dependability bottlenecks (Hsueh, Iyer, and Tsai 1997). Subsequent fault removal utilizes the information obtained via testing to improve dependability, by fixing bugs, redesigning processes, or even altering specifications to accord with new insights. Though generally thought of as an offline task, testing can also be performed during online operation, to promote peace-of-mind that all systems are functioning properly, or to provide an early-warning mechanism to

detect faults before they lead to failure. Likewise, fault removal can take place in a live system, by masking identified faults and potentially designating them for repair (Hsueh, Iyer, and Tsai 1997).

3.3.6 Deployment

Once a system is deployed into its live operating environment, techniques for preserving its correct functionality enter the realm of fault tolerance. All of the other techniques—prevention, forecasting, injection, and removal—can be seen as attempts to minimize the need for FT, while effective FT can be viewed as the culmination of everything learned about the system using those techniques. Targeting FT strategies where the least additional investment yields the greatest dependability gains not only makes sense in terms of efficient resource allocation, but also minimizes the risk of compromising those gains as a result of the additional burden of FT on the system. This architectural balance is somewhat akin to construction of a building—the infrastructure must be reinforced so that it can withstand anticipated threats without collapsing under its own weight. Of course, deployment is not the end of a project, it is merely another beginning. Criteria for evaluating the successful performance of the system must be established and measured to provide feedback for maintenance and enhancement over its full lifetime (Chillarege et al. 1993).

3.4 Techniques for Software Fault Tolerance

Practitioners of IEM generally lack the resources to specify complete systems in order to meet their goals. More commonly, computer musicians utilize general-purpose hardware and operating system platforms that can be somewhat customized to meet their requirements of dependable real-time responsiveness. COTS computers and operating systems are primarily designed to support the broadest possible variety of potential applications, design priorities which are often not aligned with maximum dependability or real-time predictability. As a result, even highly accomplished IEM programmers using COTS subsystems must rely on strategies implemented purely at the application level to achieve their objectives.

One of the most accessible and effective techniques for FT is redundancy. Redundancy tends to be an efficient means to high dependability, because it is usually easier to deploy a backup component than to engineer another order of magnitude of reliability into the original. For example, if a file server were rated at 99% reliability over the course of one year, it would be forecasted to be out of service for nearly 88 hours during that period. Simply adding a duplicate backup brings reliability up to 99.99%, with a forecasted downtime of just under one hour per year, without the effort of isolating the fault characteristics of the server and redesigning it from scratch. An even better approach might be to use different models or even servers from different manufacturers with comparable reliability, to minimize the likelihood of cascading failure. Such dramatic improvements can be made throughout a system using either redundant COTS hardware

or software redundancy techniques, without significant alterations to the underlying operating environment (Gray 1985).

3.4.1 Time Redundancy

Software redundancy can take two approaches: time redundancy or space redundancy. In many environments, the majority of faults are temporary (Gray 1985), so a faulty operation can be 'shifted in time' and repeated using the original inputs in the hope that the fault will not reappear (Heimerdinger and Weinstock 1992), a process known as rollback (Avizienis, Laprie, and Randell 2001). In contexts where integrity is paramount, such as databases and server filesystems, each collective operation is considered a transaction, and each transaction is coordinated in order to enable rollback by meeting criteria described in shorthand as ACID—atomicity, consistency, integrity, durability:

- Atomicity: Either all or none of the actions of the transaction should "happen". Either it commits or aborts.
- Consistency: Each transaction should see a correct picture of the state, even if concurrent transactions are updating the state.
- Integrity: The transaction should be a correct state transformation.
- Durability: Once a transaction commits, all its effects must be preserved, even if there is a failure. (Gray 1985)

By conceiving of system state changes as transactions and designing software that satisfies the ACID standards for each transaction, the system can be verified at each step, and upon detection of a fault, can be rolled back to the last known correct state and subsequently rolled forward again. A related concept is the recovery block (RB), in which the output of a subroutine is subjected to an acceptance test; a failed test results in

re-initialization and re-execution of the RB or an equivalent backup RB (Arlat, Kanoun, and Laprie 1996). Combining the strengths of redundancy and diversity yields n-version programming (NVP). NVP involves developing equivalent modules that are interchangeable on the 'outside', providing the same input and output interfaces, but different on the 'inside', in terms of how the routines are actually programmed. When a fault is detected and the related transaction is rolled back, an NVP variant is selected for re-execution, minimizing the risk of triggering the same fault (Xu and Randell 1997). These time-redundant techniques can yield very effective FT, but they rely on the luxury of allowing operations to start over and try again, which may not be compatible with real-time pursuits such as IEM.

3.4.2 Space Redundancy

Space redundancy relies on duplicate resources, functions, and/or data, to provide multiple simultaneous operational pathways. This type of redundancy is the most effective means of tolerating permanent faults, and the most viable option for real-time applications. In software, dual parallel processes usually provide adequate FT coverage to shift dependability bottlenecks to other areas of the system, so these redundant functions are known as process-pairs, with several approaches to implementation. In lockstep, the most basic design, primary and backup processes synchronously execute identical operations on the same data, with the disadvantage that a fault in either instructions or input could disrupt both processes in the same way. Diversity through NVP can be applied to lockstep processes to improve dependability, one situation in which several

parallels can be useful. In cases where multiple processes must return a unified result, some means of arbitration must take place, especially in the instance of NVP processes in which subtle differences in implementation could yield unpredictable behavior upon failure. For example, one indication of a fault would be value disagreement between lockstep processes. With only two choices, determining the correct value could be difficult. In some situations, averaging the two values might be acceptable. If not, three or more choices enables a vote on the correct value, assuming a consensus can be reached. In more sophisticated scenarios, in which some NVP variants might be more proven than others, this vote can be weighted to fine-tune confidence in the result, while potentially maintaining real-time performance (Heimerdinger and Weinstock 1992).

Processes can also be paired sequentially, in a primary and backup relationship. A very straightforward example is known as persistence, in which the backup process knows nothing about the primary process until it is called to take over, thereby avoiding the fault responsible for failure of the primary process. This amnesia would seem to be an insurmountable obstacle, except in combination with stored transactions as described above, in which case the backup process could merely check the completion state of the last transaction on record, roll it back if necessary to ensure system integrity, and proceed normally. Though simple and effective, persistence still carries a temporal penalty that could introduce timing faults during system normalization. A viable real-time alternative is checkpointing. In state checkpointing, the primary process communicates its state after each operation to the backup process, which remains dormant unless the primary process fails. State checkpointing provides good FT, but can be I/O intensive and difficult to

program. Delta checkpointing reduces I/O by only communicating changed state information (Gray 1985). For example, during manipulation of a multidimensional array, it is far more efficient to send only updated cells instead of the entire contents of the array at each checkpoint.

3.4.3 Failover

These sequential process-pairing techniques rely on timely failover from primary to backup process, usually accomplished by way of a heartbeat or watchdog timer (Gray 1985). A heartbeat from the primary process is merely a brief message to the dormant backup process indicating proper functioning. Upon detection of an unrecoverable fault or silent termination of the primary process, the heartbeat stops. After a predefined period of time in which it does not receive a heartbeat from the primary process, the backup process wakes up and becomes the primary process, sending its own heartbeat signal. If the other process is later repaired or comes back online, it defaults to the backup role until the heartbeat is interrupted, and so on. A watchdog timer is very similar, except the communication is not directly between the primary and backup process but with a third timing process instead. The timing process acts as a falling flag—if it is not raised (i.e. a bit set to 1) at regular intervals by the process A, it drops (i.e. changes to 0). Process B polls the flag, wakes up if it has dropped, notifies the timer that it has taken over, and the timing process subsequently becomes a rising flag that must be held down by process B. A watchdog design is most useful with distributed systems, in which other processes may need to reconfigure depending on which process is currently primary. For example, if

processes A and B run on separate machines with different network addresses, a control program concerned about network traffic volume would only send to the primary process, but would need to change its target destination if a failover occurred.

The most crucial requirement of any process-pairing design is access to a dependable clock source. No hardware resource has a greater impact on FT mechanisms, so an accurate and fault-tolerant system-wide clock service is a foundation of any good FT design (Heimerdinger and Weinstock 1992). This guideline is particularly pertinent for real-time applications, not only for its obvious importance to precise scheduling, but for prioritizing FT response measures. After fault detection, a real-time fault repair algorithm must be able to determine how much time is available to make corrections in order to choose the best course of action. Effective FT can be crucial in such time-sensitive applications, because often the time-scales of fault detection and repair are far too small for adequate human intervention.

3.4.4 Time-Triggered and Event-Triggered Systems

Real-time systems can be characterized as time-triggered, event-triggered, or a combination thereof. A time-triggered (TT) system follows a predetermined absolute timeline—when the clock reaches tick X, function Y occurs (Kopetz 1994). The clock can follow external real-world time, an internal stopwatch time initiated at process launch, or even some artificial non-linear notion of time. Regardless, the clock is the single authoritative control source, and it is expected to move unidirectionally. Most

music that follows a traditional predetermined score, with measures, meter, and tempo, is TT, including many improvisational frameworks. An event-triggered (ET) system waits for input and responds accordingly—when a polled sensor returns X, function Y occurs. Sensors can indicate human interaction such as a key press or mouse click, monitor threshold conditions such as environmental temperature, or reflect the status of other software processes. The purpose of most ET musical systems is to enable the computer accompaniment to follow a performer instead of vice versa, eliminating a consistent criticism of instrument-and-tape dynamics. In hybrid systems, a clock-driven process can yield control if interrupted by an event, or an event-driven process can yield control to a clock after a certain elapsed period of time (Kopetz 1993). Most modern COTS operating systems and applications incorporate both of these hybrid approaches, leading to the semi-autonomous yet fully responsive user interfaces with which we have all grown familiar. As noted above, the most critical component of a TT system is the clock, and naturally the most critical components of an ET system are the sensors. A robust system of either type will be able to tolerate and compensate for a faulty control source (Marzullo and Wood 1991).

Event-triggered architectures often benefit from decentralization, in which a certain degree of processing capability is located in or near the sensors. Migrating intelligence towards the perimeter of a system, an extension of the modularity and specialization introduced by OOP, can be beneficial in many ways. ‘Smart’ sensors can filter noisy input or track only meaningful state changes before signaling the core process, increasing performance. Demands for memory and instruction processing can be offloaded from the

core, making more efficient use of resources. FT can be implemented locally at each sensor or array of sensors, simplifying design and thus increasing dependability. Such distributed systems, which can follow client/server (Edelstein 1994) or peer-to-peer (Ripeanu 2001) paradigms, require special consideration for effective FT. The self-contained core and satellite modules follow principles already discussed, so the primary concern is designing fault tolerant communication amongst these modules. The actual implementation would depend on the nature of the communication, whether between processes with access to the same shared memory space, or processes running on different machines separated by thousands of miles. In any real-time distributed system, however, the temporal aspects of communications are as important as those of scheduling and processing, so FT must take into account the effects of delayed and out-of-order messages in addition to lost or corrupted messages. Distributed real-time architectures can provide great power and flexibility for implementing dynamic, responsive systems with high dependability, but these advantages come at the cost of increased complexity (Stankovic 1992).

3.4.5 Human Error

The final factor with a major influence on system design and FT is human error. Like other faults, operator error is inevitable, but can be anticipated, contained, and repaired by well-designed FT. One approach is to consider the human-computer relationship as a unified organism, a single coordinated system. In such a system, the operator has certain strengths (i.e. intelligence, mobility, creativity) and weaknesses (i.e. unpredictability,

limited system status awareness, relatively slow response), and the machine has certain strengths (i.e. precise execution of operations, relative predictability, data manipulation speed) and weaknesses (i.e. rigid response heuristics, relatively brittle architecture, limited awareness of reality). Specific nature and degree of such strengths and weaknesses vary considerably across the range of humans and machines, but underlying any system are assumptions about the character of its components, and these assumptions must be explicitly exposed, examined, and understood in order for that character to be effectively anticipated. During specification and design, it makes sense to partition tasks between human and computer to emphasize respective strengths and mitigate weaknesses. Likewise, the I/O interfaces between human and computer should be optimized for efficient completion of assigned tasks. For example, the general-purpose human-computer interface (HCI) is a keyboard, mouse, and display. This is an excellent HCI for word processing because the keyboard is optimized for language manipulation, the mouse is forgiving of imprecise motor control, and the display can leverage preexisting knowledge through extensive use of visual metaphor. The standard graphical user interface (GUI), the software component of the hardware HCI, is reasonably easy to learn and use, and is comforting to the average user because it usually reflects change only in response to explicit user commands, reflecting a dominant event-triggered model. However, the standard HCI and GUI can impose a terrible handicap against virtuosic manipulation of multi-dimensional non-linguistic abstractions in real-time, especially for time-triggered models, all of which are frequently required in IEM. Recognizing the impracticality of completely redefining the HCI for COTS systems, thoughtful design can at least provide some remedy for such limitations (Orio, Schnell, and Wanderley 2001).

Human error can be minimized by documentation, training, hands-on usage analysis, and other tactics to help the human adapt to how the computer functions, but equal if not greater attention should be dedicated to adapting the computer to how the human functions. Most operator error can be considered the product of inadequate design (Dekker, Fields, and Wright 1997). From an FT standpoint, the objective is to bridge the gap between design constraints (i.e. deviation from a perfect design due to budget, expertise, or technology) and human capabilities (since we can only be redesigned to a modest extent) to achieve a flawlessly performing integrated system.

3.4.6 The Dependability Paradox

A discussion of FT would not be complete without recognizing the truth in the old axiom "if it isn't broken, don't fix it," which turns out to be supported by statistical research. Systems undergoing change experience higher failure rates (Gray 1985). One strategy for dependability is to install proven hardware and software and then leave it alone. However, software seems to be constantly under revision, with updates regularly released to fix bugs in previous versions. This cycle of fault removal is undoubtedly beneficial if a system is suffering from one of the bugs in question, but also carries the risk of introducing new bugs into previously stable subsystems. These conflicting perspectives raise the central paradox of FT—any changes, including attempts to increase dependability, carry the risk of undermining their own objectives (Gray 1985). The only defense is eternal vigilance.

3.5 Music and Fault Tolerance

Though FT may seem to be strictly associated with computer-based systems, the philosophy behind it can be used as a perspective to view various forms of human endeavor for centuries, including music. While perhaps not an explicit goal at the time, the implicit advantages of FT have informed many aspects of technological innovation in the history of musical practice. For example, in the development of Western string instruments such as the lute, multiple strings not only facilitated chords and rapid scalar progressions, they also gave redundancy in the event of a broken string during performance. This capability become most famously celebrated with Paganini, who it was rumored intentionally sabotaged the strings on his violin in order to showcase his virtuosity:

> "One evening a rich gentleman begged ... Paganini and [a guitarist named] Lea, together with a cellist named Zeffrini, to serenade his lady-love ... Before beginning to play Paganini quietly tied an open penknife to his right arm. Then they commenced. Soon the E string snapped. "'That is owing to the damp air,' said the violinist, and kept on playing on the other three strings. "A few moments later the 'A' broke ... but he went on playing. Finally the 'D' snapped, and the love-sick swain began to be fearful for the success of his serenade. For what could Paganini do with only one string on his violin. But Paganini simply smiled and went on with the music with the same facility and strength of tone that he had previously used on all four cords." (Gates 1895)

The evolution of the orchestra from a small chamber ensemble into large sectional groupings, with each part covered by multiple identical instruments, not only increased volume, it also increased dependability. One could even imagine the orchestra as a distributed, real-time object-oriented system. Each musician is an independent object

communicating with the other objects in her subsystem via body language and audible cues. These means of communication are not sufficient to coordinate the entire system, so the conductor provides a reliable master clock to synchronize the independent processes. Even in a professional orchestra, faults are relatively common—wrong or mistuned notes, incorrect or imprecise rhythms, and equipment or personnel problems of one sort or another. In a group like the National Symphony Orchestra, these faults are likely to be masked to an extent that the audience is not even aware of them. Each musician is self-correcting, adjusting technique or instrument on-the-fly to compensate for faults and optimize performance, with multiple sensory references available to determine the state of the system at any given moment. On the opposite end of the spectrum, in an elementary-school student orchestra, faults will probably be heard as errors by the audience, but still the system is almost guaranteed not to fail entirely. The music may not receive the best treatment, but the essence will still be projected, because of the FT inherent in the architecture of the orchestral system. The dependability of the modern orchestra has conditioned audiences into expecting the same degree of FT in any musical performance, a high standard indeed.

The traditional composition process can also be seen as inherently following the dependability-explicit development model in many ways. One might go so far as to conceive of an orchestra as the hardware, and the score—instructions to be performed by the hardware—as the software. From this perspective, the characteristics and limitations of the system are well-known and well-documented, having evolved over the course of centuries, leading to informed specification and design. The composer has limited options

for changing the system structure or behavior, but an orchestra is so flexible and versatile that it can accommodate a vast range of intention without significant alteration to the system. Likewise, the orchestra is sufficiently predictable that prototyping and iterative development are largely unnecessary. An experienced composer can trust her mind's ear, refined through extensive listening and knowledge of repertoire, to know how her instructions will be translated into sound (Kennan and Grantham 1997). Rehearsal is the testing phase, in which individuals optimize their own delivery of service, and then integrate into the operation of the group as a whole. Finally, deployment—a concert—takes place, and with sufficient FT, achieves a transcendent synergy.

If we conflate the traditional orchestral environment with digital signal processing (DSP) technology, we might imagine a future optimized for dependable IEM. Aspiring musicians would choose a computer in their youth and learn its possibilities, complexities, and idiosyncrasies in exhaustive depth through years and decades of intensive study. A community of specialized craftsmen would build and maintain a dozen different computer models, each narrowly configured and optimized for a specific function. These computers would evolve over time with deliberate slowness, as composers patiently explored and exploited their unique characteristics in the search for new sounds and expressive potential. While this vision is obviously a parody, it actually comes remarkably close to describing the first generation of electronic music, as the community of innovators struggled to assimilate new technology into the prevailing musical paradigm. Populist market forces eventually took over, promoting hybrid instruments such as the electric guitar and synthesizer, but the more profound revolution

occurred when personal computers became sufficiently powerful to emulate any other instrument. The allure of versatility, the potential of a single tool to respond to any gesture and generate any sound, underlies the dream of electronic music (Varèse 1966). In contrast to the traditional instrumental virtues of stasis and specialization, the tools of IEM are continuously and rapidly evolving, an ecosystem of digital organisms offering up new and engaging possibilities with every short-lived generation (Milicevic 1996). For an experimental musician, this new environment is intoxicating, the desire to stun an audience with something they have never seen before almost overpowering. However, it should be clear that the very qualities that make IEM so interesting are the same qualities that make it unreliable. FT can reconcile these opposing imperatives of innovation and dependability.

3.6 A General Model for Interactive Electronic Music

IEM includes a vast range of hardware and software configurations, but most of these share common attributes that can be usefully categorized for the purpose of analysis and application of FT. Taxonomies of computer music practice and pursuit have been proposed and discussed (Spiegel 1992, Pope 1994), but rather than addressing each niche independently, the approach here will be to define a broadly applicable general model and then use a specific common example to elucidate that model. By definition, IEM is a real-time activity. Whether by triggering samples, driving a synthesis engine, or controlling filter parameters, live interactive music produces sound that derives from input during the performance. This responsiveness distinguishes IEM from tape music, in

which a prerecorded artifact is presented without any possibility of influence by the performer. There is some blurring between these categories in the form of acousmatic music, which is diffused through a listening space by treating the mixer and loudspeakers as an instrument, but while many practitioners take live multichannel spatialisation very seriously, most do not consider it IEM (Chadabe 1997, 131). Computer music often takes advantage of modularity and networking, with multiple embedded microcontrollers for input diversity, or CPU clusters for increased processing power, resulting in distributed systems that require consideration of additional issues beyond those of monolithic systems. IEM is also increasingly combined with visual elements that are interactively generated, either algorithmically or in collaboration with a video performer. This kind of intermedia work adds another dimension of artistry and sensory engagement for the audience, as well as additional complexity to the supporting electronic systems, but from an FT point of view it is still just a collection of real-time processes that must be synchronized, and the same principles apply to live experimental video (LEV) as IEM.

Functionally, the basic IEM system includes input, processing, and output. More specifically, input arrives via sensors as control or audio sources, which send and potentially receive data to/from other subsystems using some physical medium and communications protocol. One subsystem coordinates this I/O and will be considered the CPU, although it may be distributed amongst multiple pieces of hardware. The CPU analyzes inputs, and according to programmed instructions, maps them to outputs as control feedback and/or audio parameters. These outputs are transmitted in turn to their pertinent subsystems as an updated representation of system state, via control abstraction

and/or analog sound, respectively. FT can be applied to each subsystem and interaction between them—sensors, communications, CPU, and the digital-to-analog pathway from software to the audience.

In my experience as a composer, performer, and consumer of computer music, having witnessed over one hundred concerts with at least a few hundred IEM works, I have seen many different implementations of this general model. However, strong commonalities among these diverse systems emerge. Inputs often include microphones, which convert sound as air pressure into continuously variable voltage, which is subsequently sampled by an analog-to-digital converter (ADC) at a certain frequency and bit depth such as the compact disc (CD) 44.1 KHz/16-bit standard to produce discrete values. Another common input is a tactile sensor, which translates physical motion via a switch, rotary or linear potentiometer, or force-sensitive resistor (FSR) into variable voltage. Environmental sensors, such as those for temperature, light, or electromagnetism, and hybrid sensors such as those for acceleration and tilt also operate on the same principles. These sensors usually pass through an ADC into a 7-bit (0–127) or 8-bit (0–255) range of values, updated upon change or at a control rate considerably slower than the audio rate. Communication is largely bandwidth-dependent. Post-ADC, digital audio communications tend to use standards defined for the purpose, such as S/PDIF (IEC 1999) or AES/EBU (AES 1985) for stereo data and Alesis ADAT or Tascam TDIF for multi-channel data. In the case of COTS personal computers, ADCs are likely to reside in the same box as the CPU, in which case digital audio data is transferred via Peripheral Component Interconnect (PCI) or a similarly integrated high-throughput host bus.

Control communications often use MIDI (MMA 1996) or other serial protocols, and network-based transports such as OpenSoundControl (Wright 1998) via UDP over Ethernet are becoming more widely used for this purpose. There are only a handful of probable CPU types—those compatible with the x86 instruction set (Intel, AMD and Transmeta), the PowerPC (IBM and Motorola), and conceivably the Sun SPARC. Widening the definition of CPU a bit would also bring in dedicated Digital Signal Processing (DSP) chips such as those in a Symbolic Sound Capybara (currently Motorola DSP-56309), those on a Digidesign HD Process card (currently Motorola DSP-56301), and others similarly used to offload audio processing from a host CPU. Likewise, there are only a few mainstream operating systems: Linux and its Unix-based brethren, two versions of MacOS, and Windows. The multiplicity of real-time musical software available for these architectures is far too rich to enumerate here, but for IEM driven by sensors beyond a keyboard and mouse, a few applications do stand out. Csound, jMax, Pd, MAX/MSP (Zicarelli 1998), SuperCollider, and RTcmix are all frequently used for IEM, and all are more accurately described as development environments, because each provides a framework and set of tools that enable a composer/programmer to customize and extend the software in virtually every respect. Returning from the application level to the audience reverses the flow—back through the operating system, the CPU, communications pathways, and into the signal amplification subsystem. There are a vast number of potential variables at this stage, but for the purposes of FT it will suffice to say that almost all sound reinforcement involves digital-to-analog converters (DAC), mixers, outboard DSP (effects) modules, amplifiers, and loudspeakers. As mixers become more sophisticated, combining DSP with their more traditional functions of regulating signal

amplitude, equalization, and routing, they can be considered another CPU, complete with operating system and possible other software. Finally, we shouldn't forget the human performer and her acoustic instrument as additional subsystems with their own considerations. This formidable array of components and interconnections, seemingly increasing in complexity with each successive generation, represents the scope of the challenge in applying fault tolerance to interactive music.

3.7 Case Study

To make the task of integrating FT into IEM more manageable, I have selected a representative system from all the possibilities enumerated above and implemented a case study. This demonstration makes explicit the connection between theory and practice, describing the problems one might encounter, and justifying the solutions I would choose to address them, based on the principles and techniques discussed thus far. In an effort to balance minimum expenditure of effort and resources with maximum dependability gains, the first step is to determine where failures are most likely to occur, and where the composer/programmer has the most power to apply FT. Every situation is unique to some extent, but based on my own informal observations, I will make some generalizations about the dependability of the various components in IEM:

- **Performer**. Dependability: Moderate. Fault prevention is the key here. Choose a performer the way you would choose a pilot for your next airplane flight—for skill, professionalism, and consistency. If partway into the project the performer

seems to be the dominant cause of failures, fault removal might be in order (be diplomatic—humans are more sensitive than machines about this sort of thing). Adequate rehearsal is critical. Ideally, the performer becomes an ally in FT, able to mask faults elsewhere in the system via improvisation and stagecraft.

- **Acoustic instrument**. Dependability: High. A well-maintained, high-quality instrument rarely fails except under extreme stress. The idiosyncrasies of most instruments are well-documented, so familiarity with the literature and the experience of a performer-collaborator facilitate accurate fault forecasting. Fault tolerance relies on either the redundancy of a duplicate instrument close at hand or the ability of a performer to mask faults by adjusting technique.
- **Control sensors**. Dependability: Moderate. Sensors may be built to industrial-strength specifications or be cobbled together with duct tape and super glue, so it is up to the system developer to assess the fault profile of the sensors and design accordingly. COTS sensors tend to be more reliable than experimental or handmade versions, primarily due to differences in the scale of testing, subsequent fault removal, and standardization.
- **Microphones**. Dependability: High. Commercial stage mics are specifically made for durability in a performance context, and as a result are extremely reliable within their specified environmental parameters.
- **ADC**. Dependability: High. Outboard ADCs are usually solid-state devices designed for a single purpose, and as such show excellent reliability given correct input. ADCs built-in to COTS computers tend to have lower fidelity, but are

otherwise equally reliable. The most likely fault is damage to a connection mechanism.

- **Communications**. Dependability: Low. In this discussion, I will include all cables, connectors, plugs, and jacks, as well as the signals and protocols that travel over these interfaces, within the realm of communications. These components are frequent causes of faults and failures, and warrant particular attention for FT.
- **CPU**. Dependability: High. I will use a broad concept of CPU here, referring not only to the processor, but also its supporting infrastructure, including motherboard, random-access memory (RAM), hard disk, power supply, and remaining hardware "inside the box." In many discussions of FT, these components are addressed independently, but in the context of COTS computers in IEM, applying additional FT at the subcomponent level would be impractically difficult, so considering the internal hardware as a monolithic system should be sufficient. Hardware from reputable vendors tends to be very reliable, with an MTBF measured in years, so if it works during development and testing, barring reconfiguration or environmental stress, it is highly likely to keep working through a performance.
- **OS**. Dependability: Low. The demands of IEM can reveal faults even in operating systems that are very stable in general use, such as a properly configured version of Linux, BSD-based Unix, or MacOS X, with uptime commonly measured in months. Less robust OSes such as Windows or MacOS 9 and earlier require particular scrutiny to optimizing stability and minimizing superfluous processes.

- **Applications**. Dependability: Low. In COTS systems, it is difficult to disentangle the fault contribution between OS and applications. A bug-free program stressing a weak aspect of an OS may trigger more faults than a poorly written program taxing a robust portion of the OS. Also, most applications are chosen to provide a specific function, with few if any interchangeable alternatives. These factors place the burden of dependability on fault forecasting and fault tolerance.
- **DAC**. Dependability: High. The DAC differs from the ADC primarily in reversal of input and output. Because digital input must follow a fixed protocol in order to be properly interpreted, it tends to have more potential for disruption than analog input. However, DACs are still usually just as reliable as ADCs. It's worth noting that both functions are often incorporated in the same subsystem, so loss of one could incur simultaneous loss of the other as well.
- **Amplification**. Dependability: Moderate. Each venue is different, making this last link in the chain of performance very difficult to assess. The configuration is largely beyond control of the composer and performer, so house technical crew and their systems become an unknown factor in the dependability equation. Fault prevention may not be possible, a brief rehearsal window may not be sufficient for effective fault forecasting, and there may be no opportunities for fault removal or means of implementing fault tolerance. In this discouraging scenario, the only comfort lies in the knowledge that amplification is among the most standardized elements in IEM. If the rest of the work is properly prepared beforehand so that it requires minimal attention, sufficient opportunity should remain to grasp the

> essentials of most amplification subsystems. However, if someone pushes the wrong button on a highly-automated software-configurable mixer, all bets are off.

Fault prevention should always be practiced by making an effort to choose dependable components when possible, and all component functions should be thoroughly tested. The ratings above assume application of these principles in all cases. A dependability rating of high means I have never seen this component fail during performance. A rating of moderate means I have seen this component fail during performance but consider it unusual. A rating of low means I have frequently seen this component fail during performance. Of course, the opinions of others may vary, and I expect that readers with more experience than I will follow their own judgement.

Based upon these ratings, I chose to focus the case study on communications and OS/application subsystems, in an attempt to raise their dependability to a level comparable to the rest of the IEM components. I used COTS hardware and software because it imposes the restrictions of optimizing products provided by other parties. Custom solutions can follow the complete dependability-explicit development cycle without compromise, and it is precisely the compromises that many IEM practitioners face that I wished to address by example. In a COTS context, fault prevention lies solely in choosing which products to use. Fault forecasting can be applied, but in the absence of any power to change the product or substitute equivalent choices, there is no option of fault removal. Thus, in a COTS environment, often the only remaining approach is to

gather as much reliable information as possible and leverage that knowledge towards effective fault tolerance.

In my IEM experience, mainly in the academic and experimental communities, the predominant software environment is the family made up of MAX/MSP, Pd and jMax. As these three variants represent interpretations of the same model, I will refer to them collectively as MAX (Puckette 2002). MAX is an object-oriented graphical development tool for manipulating numerical and audio data, and lends itself particularly well to real-time, event-triggered processing, the primary paradigm for IEM. The relationship of influence between the predilections of the software and the aesthetics of the music are subject to debate (Lyon 2002), but MAX also makes an excellent platform for exploration of software FT. My fondness for, expertise with, and current use of MAX and its underlying PowerPC/MacOS-based platform complete the reasons why I have chosen MAX as the central example for analysis. I encourage readers with other preferred environments to adapt the strategies described here for their own use and to share their findings with the community.

The configuration I contrived to simulate common IEM practice included a physical interface used to trigger sample playback, control real-time synthesis, and manipulate live filtering parameters, basic staples of IEM. The CPU was an Apple Macintosh PowerBook 2400, upgraded with a 320 MHz G3 (PPC 750) processor, 80 MB RAM, and 30 GB hard disk. The operating system was MacOS 9.1, and the application was MAX 3.6.2/MSP 1.7.2. The sensor interface was a Roland PAD-5 percussion controller, with pressure-

sensitive trigger data transmitted to the CPU via MIDI. The MIDI interface was a MidiMan MacMan attached via the RS-422 DIN-8 serial port. Audio output was via the PowerBook's built-in 1/8" stereo jack. The expected dependability weaknesses in this configuration were the cables, operating system, and application. The primary option available to address these weaknesses was redundancy.

The first level of redundancy was a complete secondary system, capable of performing all the tasks of the primary system. The secondary CPU was an Apple Macintosh PowerBook 3400c, with a 240 MHz PPC 601 processor, 144 MB RAM, and 6 GB hard disk. The operating system and applications were the same. The MIDI interface was an Opcode MIDI Translator II attached via the serial port. Audio output was also via the PowerBook's built-in 1/8" stereo jack. This secondary system was chosen from convenience, but also represented a high degree of diversity. I considered installing different versions of OS and applications to maximize this diversity advantage, but my experience suggested that significantly different versions of either would result in noticeably reduced stability.

The I/O of both machines was parallelized as much as possible. The MIDI output from the PAD-5 was split through a MidiMan Merge 2x2 box, which provided duplicate independent output to both laptop interfaces from the single input. Audio output was patched independently from each computer into the mixer on separate channels, and levels were normalized so that a transition from one machine to the other would not alter the character of the audio. This configuration also allowed muting either machine in case

of a malfunction resulting in unwanted noise. Although no microphones were used in this case study, parallelizing audio input to each computer could have been easily implemented if necessary. Communications between the two machines was accomplished using TCP/IP over the IEEE 802.11b wireless networking protocol (WiFi), with Lucent Orinoco Gold PCMCIA expansion cards in peer-to-peer mode. While the bandwidth of WiFi is significantly less than wired Ethernet, this limitation was not prohibitive. In the demo, the computers would be close and in line-of-sight to each other, so I decided to compare the dependability of WiFi to the known fault potential of yet another cable.

The redundancy described thus far could provide high availability with respect to most hardware and software faults. For many IEM scenarios, it might be acceptable for both machines to generate audio output simultaneously—one active and the other muted at the mixer—with someone poised to swap channels in the event of a failure, a form of manual fault masking. For demonstration purposes, however, I decided it would be more interesting to implement an automatic failover mechanism, so that the FT could perform as autonomously as possible. The main requirements for such a mechanism were fault detection and fault compensation—sensing a critical problem and invoking the other computer to take over operation. I decided that automated fault diagnosis would be impractically complex, and manual fault diagnosis useless in a real-time context, but aspects of fault containment, fault masking, and fault repair were found to be feasible and useful enhancements to dependability.

Support for these requirements had to be fully integrated into the application, so I naturally followed a dependability-explicit development process within the MAX environment. The specification was simple: to create a system that would enable real-time physical control of sample playback, real-time synthesis, and live filtering parameters, able to tolerate complete failure of any subsystem with less than one second of disruption. In a real IEM piece, the specification would also encompass the musical score, requiring notation sufficient to accurately describe the behavior of the system and its interaction with a performer, a challenging problem unto itself (Schedel 1999). The hardware design is described above—redundancy of all components except for the controller and the MIDI splitter. The software design became quite complicated by comparison.

As discussed earlier, the real-time nature of IEM eliminates many options for software redundancy, leaving checkpointed process-pairs as the primary means of achieving FT without introducing timing faults. The specifications of the case study mandated a hybrid architecture, including both time-triggered and event-triggered processing. Failover between two machines suggested a heartbeat monitor. For any of these strategies to work, a reliable clock was necessary, which fortunately could be delivered by the hardware, operating system, and MAX environment to the FT architecture. The desired behavior of the FT subsystem was to operate in either active or standby roles, as appropriate, without requiring different software for each. The active computer would transmit state information to its partner, including a heartbeat and necessary delta checkpoint data, while the standby computer would not generate any audio unless a predetermined time

elapsed after receiving the last heartbeat, at which point it would switch to the active role. As noted earlier, WiFi adapters were dedicated to FT-related communications.

The structure of the IEM techniques followed a modular approach—each function operated in sequence, with a means of stepping from one to the next. The first configuration triggered sample playback, the next controlled real-time synthesis, and the last manipulated live filtering parameters. Compared to most real works of IEM, this form was artificially didactic, in order to be clearly illustrative, but the same structure could be used to support concurrent functions without any major alteration. The behavior of the IEM subsystem was straightforward. The patch initialized to 'Config 0', which enabled silent testing for proper communications—taps on the pad controller would trigger visual feedback but no sound. A double-tap on pad #5 advanced to 'Config 1', in which each pad triggered a specific drum kit sample. Another double-tap on pad #5 advanced to 'Config 2', in which a tap on any pad triggered a tone generated by FM synthesis, in a rising scale pattern. Another double-tap on pad #5 advanced to 'Config 3', which started playback of a sound file, with intensity of filter processing directed by the velocity of subsequent taps on any pad. A final double-tap on pad #5 quickly faded the audio output to zero, ending the demonstration. In order to compensate for the possible failure of the pad sensors, the computer keyboard could also be used to trigger the same events, using the numerals 1 through 5 to exhibit fault masking. Though sacrificing the dimension of velocity control, this graceful degradation represented a reasonable contingency in the absence of a second PAD-5.

The communication required to enable seamless failover between machines was also straightforward: a heartbeat signal, and patch state information. The heartbeat was the briefest message possible, in order to preserve bandwidth, sent at an interval of 500 ms. The state information included only the minimum amount of data necessary for the standby to pick up where the primary left off, starting with the Config number. 'Config 1', the most basic possibility, functioned without maintaining any memory or context from one event to the next. 'Config 2' required sending the scale degree associated with each tap so that the next would proceed in the proper sequence. 'Config 3' not only had to mirror the velocity of each tap, but the playback position within the sound file in real time. These three levels of complexity corresponded respectively to the event-triggered, time-triggered, and hybrid triggering models representative of IEM practice.

From this design, based on the specifications and resources available, the details of development and implementation emerged. The choices supporting FT did not dictate a particular programming method or style, but instead required an integration layer, linking each significant state representation to the communications interface. This FT layer listened to everything important happening in the system, in preparation to act in case of emergency. However, rather than handling all input at the periphery, it only duplicated essential status changes. For example, the PAD-5 transmits a constant stream of running status, which is suppressed by MAX as superfluous data and never sent to the standby. Likewise, a key press other than 1, 2, 3, 4, or 5 would not reach the FT layer. By mirroring events only after they have reached a point of significance, not only is bandwidth used most efficiently, but the standby is insulated from possible ill effects of

spurious influence, enhancing dependability. The FT layer also imposed fault containment by validating outgoing and incoming data with range limits and continuity checks where appropriate. If a value error was detected, it was either truncated to the nearest acceptable boundary or constrained using a best-guess algorithm, depending on the disruptive potential of the error. For example, if the playback position value in 'Config 3' jumped backwards, or forwards by a large margin, it was interpolated based on the last known good value instead, to prevent expression of an obvious discontinuity. Such basic sanity checks can be very effective at inhibiting propagation of faults to failures. The processing overhead of this entire FT layer was nominal compared to the demands of real-time DSP, and did not seem to destabilize the system in any way.

I presented this case study at the SEAMUS 2001 conference in Baton Rouge, LA, so it was treated as a performance, and rehearsed accordingly as part of the testing regimen. Though I did not quite practice my technique for hours per day as many professional musicians do, that same level of dedication to a computer-based instrument does not seem at all unreasonable. Whenever a failure occurred in rehearsal, the responsible fault was isolated and repaired if possible. If not, the system was revised to enable continuing the demonstration despite the recurrence of such a failure. Of course, some breakdowns were an intentional part of the presentation. During each 'Config', the primary computer was forced to fail, in order to illustrate seamless recovery by the standby. Then the failed computer was brought back online as the new standby—relaunching the application or silently rebooting as an example of fault repair—during explanation of the next 'Config'. While in this case the FT components represented the actual work created for the

audience, ordinarily in IEM the FT would operate behind the scenes. However, in an FT-supported piece, the FT subsystems should be tested just as thoroughly as the main music-related routines, by systematically injecting faults at all levels before and after the FT has been engaged.

Deployment is the final hurdle, and on such occasions it is wise to remember the Boy Scout motto: Be Prepared. For my SEAMUS presentation, I had a special bag containing every kind of cable and adapter I might conceivably need. I had backup batteries, backup AC adapters, and backup CD-ROMs containing the OS and applications so that I could restore each computer to its operational state in a matter of minutes if necessary. Depending on the software, it is often even possible to create a completely bootable performance platform on CD or DVD, so that spare computers at the venue can be called into service at a moment's notice. Fortunately, I did not encounter any major issues during the demonstration. As the system was progressively degraded, each 'Config' module failed over with only the slightest interruption—barely noticeable even when everyone was cued to listen for it and could see it projected on the screens behind me. Eventually, the PAD-5 was disconnected, various cables were unplugged, and at the end, both laptops were conspicuously disabled, but the piece kept on playing. At the beginning of 'Config 3', I had surreptitiously started a portable MP3 player with the same sound file as on the computers at that point. After the final laptop failure, I quickly swapped the audio cable over to the MP3 player while directing attention elsewhere—the audience was listening so intently to what I was saying that they didn't even notice the switch. Like magicians, shrewd practitioners of IEM should have many tricks at their disposal.

This case study has discussed just a few of the possibilities for applying FT to IEM, but has hopefully provided some insight into how FT theory can be translated into practice.

3.8 Conclusion

In this paper, I have attempted to show how the principles of fault tolerance can be adapted from a computing science context and applied to improving the experience of interactive music, and achieving the artistic goals embodied therein. Learning to view IEM from an FT perspective and taking a systematic approach to a creative process requires an investment of time and energy, and commitment to a structured development framework, but the rewards can be significant. On the other hand, it seems that time is usually the scarcest resource in any project. While integrating FT into IEM may provide invaluable protection against the forces of entropy, if it compromises time that would be better devoted to a more complete artistic vision of a work, then the project might still be considered a failure despite functioning flawlessly. This is the challenge of applying a discipline founded on quantitative measurement to an artistic endeavor in which the judgement of success is infinitely more subtle. As always in IEM, balance between the technical and the creative is essential. The principles and applications of FT presented here are merely another set of tools at our command, with the distinction that they have the power to make every other tool we use more predictable and dependable. The fundamental reason to apply FT to IEM is to instill confidence in the systems supporting the creation of musical experience. When a performer trusts a computer as much as a piano, only then will the stage be set for realization of a masterpiece.

3.9 References

Audio Engineering Society. 1985. AES recommended practice for digital audio engineering—serial transmission format for linearly represented digital audio data. *Journal of the Audio Engineering Society* 33 (12): 975–84.

Arlat, J., K. Kanoun, and J.-C. Laprie. 1996. Dependability evaluation of software fault-tolerance. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS)*, 194–99. Pasadena, CA: IEEE Computer Society Press.

Avizienis, A. 1967. Design of fault-tolerant computers. In *American Federation of Information Processing Societies (AFIPS) Conference Proceedings, Fall Joint Computer Conference (FJCC)* Vol. 31, 733–43. Washington, D.C.: Thompson.

Avizienis, A., J.-C. Laprie, and B. Randell. 2001. Fundamental concepts of dependability. Research Report 01-145. Toulouse, France: Laboratory for Analysis and Architecture of Systems (LAAS), Centre National de la Recherche Scientifique (CNRS).

Avizienis, A. 1997. Toward systematic design of fault-tolerant systems. *Computer* 30 (4): 51–58.

Birman, K. P. 1993. Reliable enterprise computing systems. Ed. M. Banâtre and P. A. Lee. *Hardware and Software Architectures for Fault Tolerance*, 140–50. Heidelberg: Springer-Verlag.

Chadabe, J. 1997. *Electric sound: The past and promise of electronic music*. New Jersey: Prentice-Hall.

Chillarege, R., R. K. Iyer, J.-C. Laprie, and J. D. Musa. 1993. Field failures and reliability in operation. In *Proceedings of the 4th International Symposium on Software Reliability Engineering (ISSRE)*, 122–26. Denver, CO: IEEE Computer Society Press.

Comer, D. 1995. *Internetworking with TCP/IP*. 3d ed. Vol. 1, *Principles, protocols, and architecture*. New Jersey: Prentice-Hall.

Cristian, F., B. Dancey, and J. Dehn. 1996. Fault-tolerance in air traffic control systems. *Association for Computing Machinery (ACM) Transactions on Computer Systems* 14 (3): 265–86.

Dawson, S., F. Jahanian, T. Mitton, and T.-L. Tung. 1996. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS)*, 404–14. Sendai, Japan: IEEE Computer Society Press.

Dechelle, F., M. De Cecco, E. Maggi, and N. Schnell. 1999. jMax recent developments. In *Proceedings of the International Computer Music Conference,* 445–48. Beijing, China: ICMA.

Defense Advanced Research Projects Agency (DARPA). 1 March 2003. Organically Assured and Survivable Information System (OASIS). <http://www.tolerantsystems.org>.

Dekker, S., R. E. Fields, and P. C. Wright. 1997. Human error recontextualised. In *Proceedings of the 1st Workshop on Human Error in the Development of Systems*. Glasgow, UK: Glasgow Accident Analysis Group (GAAG), University of Glasgow.

Edelstein, H. 1994. Unraveling client/server architectures. *DBMS* 7 (5): 34–41.

Gates, W. F., ed. 1895. *Anecdotes of great musicians*. Philadelphia, PA: Presser.

Gray, J. 1985. Why do computers stop and what can be done about it? Technical Report 85.7. Cupertino, CA: Tandem Computers.

Hamming, W. R. 1950. Error detecting and error correcting codes. *Bell Systems Technical Journal* 29 (2): 147–60.

Hecht, M., H. Hecht, and E. Shokri. 2000. Adaptive fault tolerance for spacecraft. In *Proceedings of the IEEE Aerospace 2000 Conference*. Big Sky, MT: IEEE Computer Society Press.

Hecht, M., J. Agron, H. Hecht, and K. H. Kim. 1991. A distributed fault tolerant architecture for nuclear reactor and other critical process control applications. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS)*, 462–69. Montreal, Canada: IEEE Computer Society Press.

Heimerdinger, W. L., and C. B. Weinstock. 1992. A conceptual framework for system fault tolerance. Technical Report CMU/SEI-92-TR-33, ESC-TR-92-033. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Herlihy, M. P., and J. M. Wing. 1991. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems* 2 (1): 93–104.

Hsueh, M.-C., R. K. Iyer, and T. K. Tsai. 1997. Fault injection techniques and tools. *Computer* 30 (4): 75–82.

International Electrotechnical Commission (IEC). 1999. Digital audio interface—Part 1: General. Technical Report IEC 60958-1. Geneva, Switzerland: IEC.

Kaâniche, M., J.-C. Laprie, and J.-P. Blanquart. 2000. Dependability engineering of complex computing systems. In *Proceedings of the 6th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 36–46. Tokyo, Japan: IEEE Computer Society Press.

Kennan, K. and D. Grantham. 1997. The technique of orchestration. 5th Ed. New Jersey: Prentice-Hall.

Kopetz, H. 1993. Should responsive systems be event-triggered or time-triggered? *Institute of Electronic, Information, and Communications Engineers (IEICE) Transactions on Information and Systems* E76-D (11): 1325–32.

Kopetz, H. 1994. The design of fault-tolerant real-time systems. In *Proceedings of the 20th EUROMICRO Conference on Systems Architecture and Integration*, 4–9. Liverpool, UK: IEEE Computer Society Press.

Lanier, J. 2002. The complexity ceiling. Ed. J. Brockman. *The next fifty years: Science in the first half of the twenty-first century*, 216–29. New York: Vintage Books.

Laprie, J.-C., ed. 1992. *Dependability: Basic concepts and terminology*. Dependable Computing and Fault-Tolerance, vol. 5. Vienna, Austria: Springer-Verlag.

Laprie, J.-C. 1995. Dependability of computer systems: Concepts, limits, improvements. In *Proceedings of the 6th International Symposium on Software Reliability Engineering (ISSRE)*, 2–11. Toulouse, France: IEEE Computer Society Press.

Lawrie, A.T. and C. B. Jones. 2002. Goal-diversity in the design of dependable computer-based systems. In *Proceedings of the 1st Workshop on Open Source Software Development*. Newcastle, UK: Interdisciplinary Research Collaboration in Dependability (DIRC), University of Newcastle upon Tyne.

Levendel, Y. 1994. Fault tolerance cost effectiveness. Ed. M. Banâtre and P. A. Lee. *Hardware and Software Architectures for Fault Tolerance*, 13–20. Heidelberg: Springer-Verlag.

Lyon, E. 2002. Dartmouth symposium on the future of computer music software: A panel discussion. *Computer Music Journal* 26 (4): 13–30.

Madden, T., R. B. Smith, M. Wright, and D. Wessel. 2001. Preparation for interactive live computer performance in collaboration with a symphony orchestra. In *Proceedings of the International Computer Music Conference*, 310–13. Havana, Cuba: ICMA.

Marzullo, K. and M. Wood. 1991. Making real-time reactive systems reliable. *Association for Computing Machinery (ACM) Operating Systems Review* 25 (1): 45–48.

McCartney, J. 1998. Continued evolution of the SuperCollider real time synthesis environment. In *Proceedings of the International Computer Music Conference*, 133–36. Ann Arbor, MI: ICMA.

MIDI Manufacturers Association (MMA). 1996. *The complete detailed MIDI 1.0 specification*.

Milicevic, M. 1996. Computer music and the importance of fractals, chaos, and complexity theory. In *Proceedings of the 3rd Computer Music Conference JIM96*. Paris, France: IRCAM.

Moore, E. F. and C. E. Shannon. 1956. Reliable circuits using less reliable relays. *Journal of The Franklin Institute* 262: 191–208.

Moser, L. E., P. Narasimhan, and P. M. Melliar-Smith. 1996. Object-oriented programming of complex fault-tolerant real-time systems. In *Proceedings of the IEEE 2nd International Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, 116–119. Laguna Beach, CA: IEEE Computer Society Press.

National Aeronautics and Space Administration (NASA). 1999. Advanced software fault tolerance strategies for mission critical spacecraft applications. Task 3 Interim Report: Assessment of X-38 201 Software Architecture and Software Development Approach. Houston, TX: Johnson Spaceflight Center (JSC).

Open Source Initiative (OSI). 1 March 2003. The open source definition. <http://www.opensource.org>.

Orio, N., N. Schnell, and M. M. Wanderley. 2001. Input devices for musical expression: Borrowing tools from HCI. Paper presented at workshop, New Interfaces for Musical Expression—CHI 2001, 1–2 April, in Seattle, WA.

Pierce, W. H. 1965. *Failure tolerant design*. New York: Academic Press.

Pope, S. T. 1994. A taxonomy of computer music. *Computer Music Journal* 18 (1).

Puckette, M. S. 1997. Pure data. In *Proceedings of the International Computer Music Conference,* 224–27. Thessaloniki, Greece: ICMA.

Puckette, M. S. 2002. Max at seventeen. *Computer Music Journal* 26 (4): 31–43.

Raymond, E. S. 1999. *The cathedral and the bazaar*. Sebastopol, CA: O'Reilly & Associates, Inc.

Ripeanu, M. 2001. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of the 1st International Conference on Peer-to-Peer Computing*, 99–100. Lingköping, Sweden: IEEE Computer Society Press.

Romanovsky, A., J. Xu, and B. Randell. 1996. Exception handling and resolution in distributed object-oriented systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS)*, 545–52. Hong Kong: IEEE Computer Society Press.

Schedel, M. 1999. The notation of interactive music: Limitations and solutions. In *Proceedings of the International Computer Music Conference*, 403–6. Beijing, China: ICMA.

Shannon, C. E. 1948. A mathematical theory of communications. *Bell Systems Technical Journal* 27 (3): 379–423, 623–56.

Siewiorek, D., and R. Swarz. 1998. *Reliable computer systems: Design and evaluation*. 3rd ed. Natick, MA: A. K. Peters.

Spiegel, L. 1992. An alternative to a standard taxonomy for electronic and computer instruments. *Computer Music Journal* 16 (3): 5–6.

Stankovic, J. A. 1992. Distributed real-time computing: The next generation. *Journal of the Society of Instrument and Control Engineers of Japan* 31 (7): 726–36.

Topper, D. 1999. RTcmix and the open source / free software model. In *Proceedings of the International Computer Music Conference*, 597–99. Beijing, China: ICMA.

Torres-Pomales, W. 2000. Software fault tolerance: A tutorial. Technical Report NASA/TM-2000-210616. Hampton, VA: NASA Langley Research Center.

Truax, B. 2001. *Acoustic communication*. Westport, CT: Ablex Publishing.

Varèse, E. 1966. The liberation of sound. *Perspectives of New Music* 5 (1): 11–19.

Vercoe, B., and D. Ellis. 1990. Real-time Csound: Software synthesis with sensing and control. In *Proceedings of the International Computer Music Conference*, 209–11. Glasgow, Scotland: ICMA.

Von Neumann, J. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. Ed. C. E. Shannon and J. McCarthy. *Automata Studies*, 43–98. Princeton, NJ: Princeton University Press.

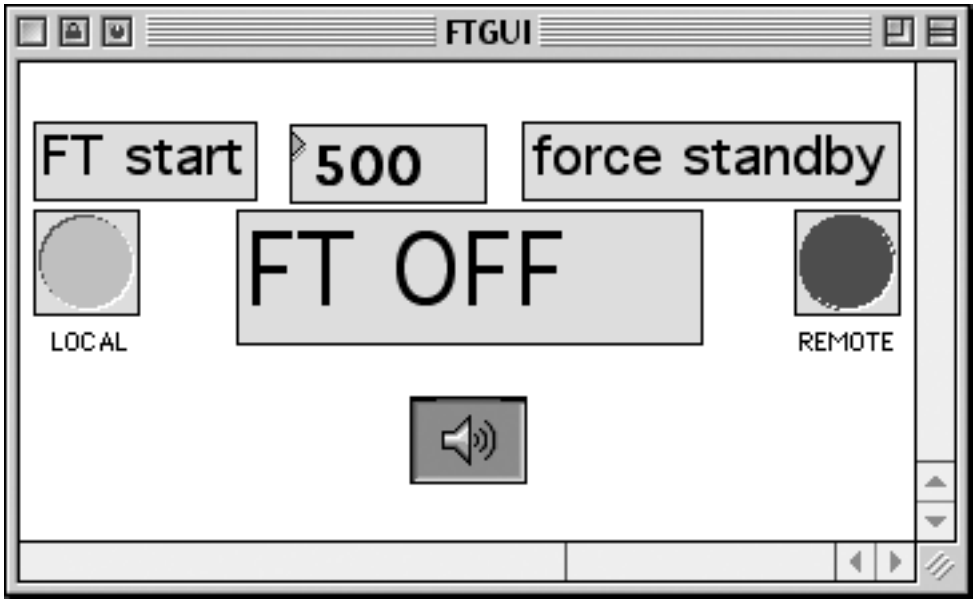
Wright, M. 1998. Implementation and performance issues with OpenSound Control. In *Proceedings of the International Computer Music Conference*, 224–27. Ann Arbor, MI: ICMA.

Xu, J., B. Randell, C. M. F. Rubira-Calsavara, and R. J. Stroud. 1995. Toward an object-oriented approach to software fault tolerance. Ed. D. R. Avresky and D. K. Pradhan. *Recent Advances in Fault-Tolerant Parallel and Distributed Systems*, 226–33. IEEE Computer Society Press.

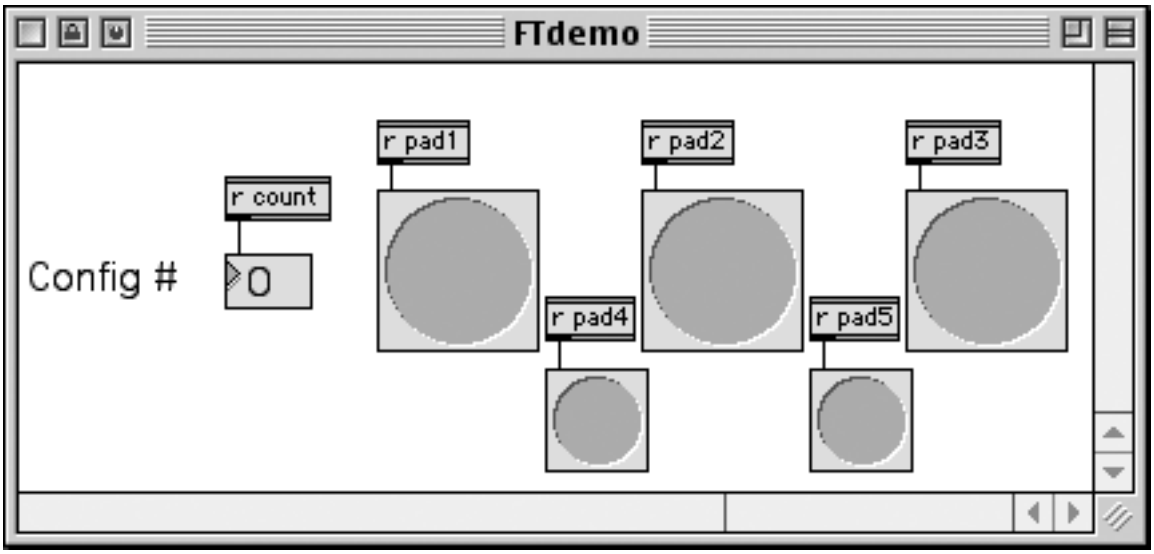
Xu, J., and B. Randell. 1997. Software fault tolerance: t/(n-1)-Variant programming. *IEEE Transactions on Reliability* 46 (1): 60–67.

Zicarelli, D. 1998. An extensible real-time signal processing environment for MAX. In *Proceedings of the International Computer Music Conference*, 463–66. Ann Arbor, MI: ICMA.

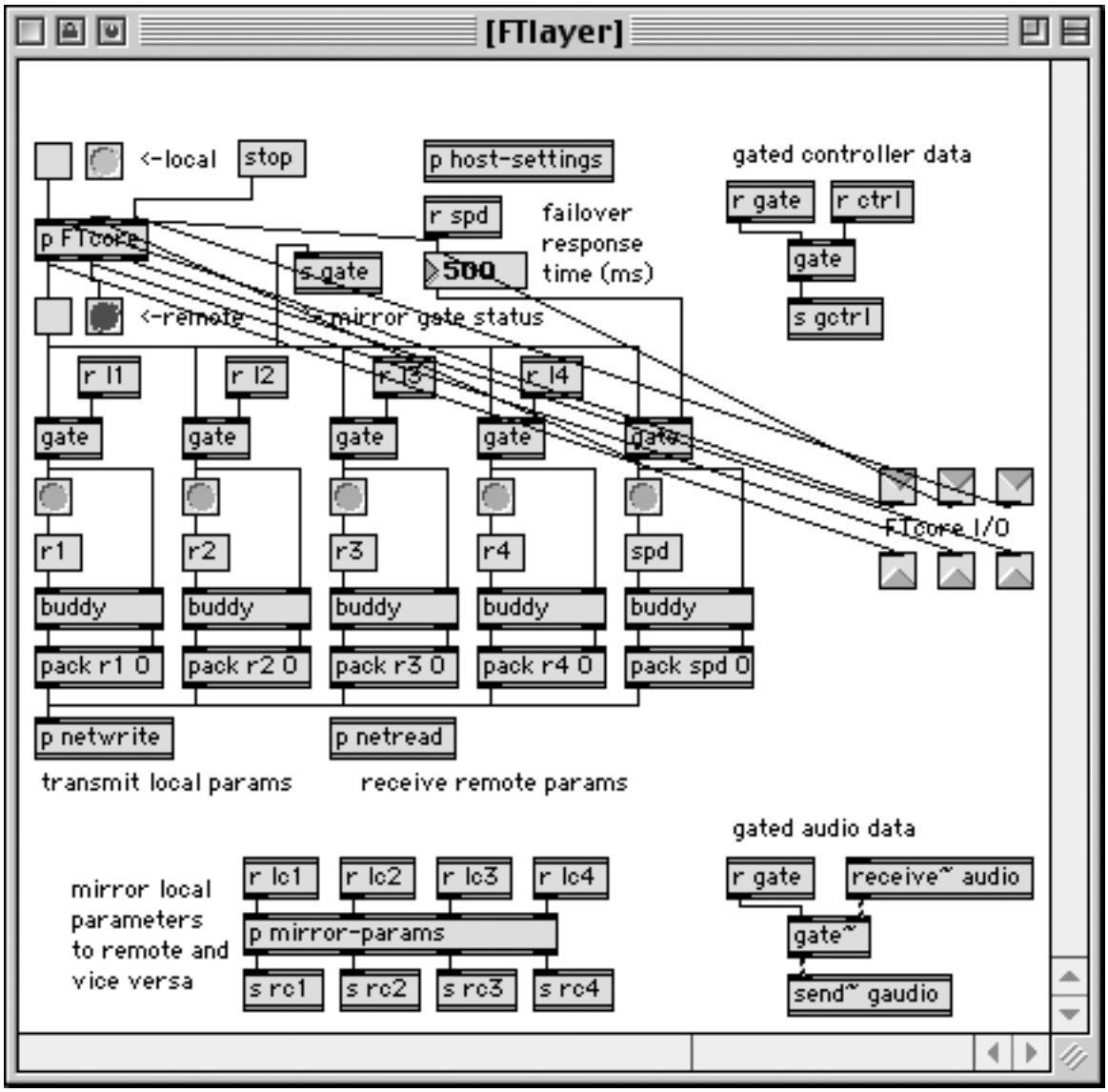
3.10 Appendix: MAX Patches



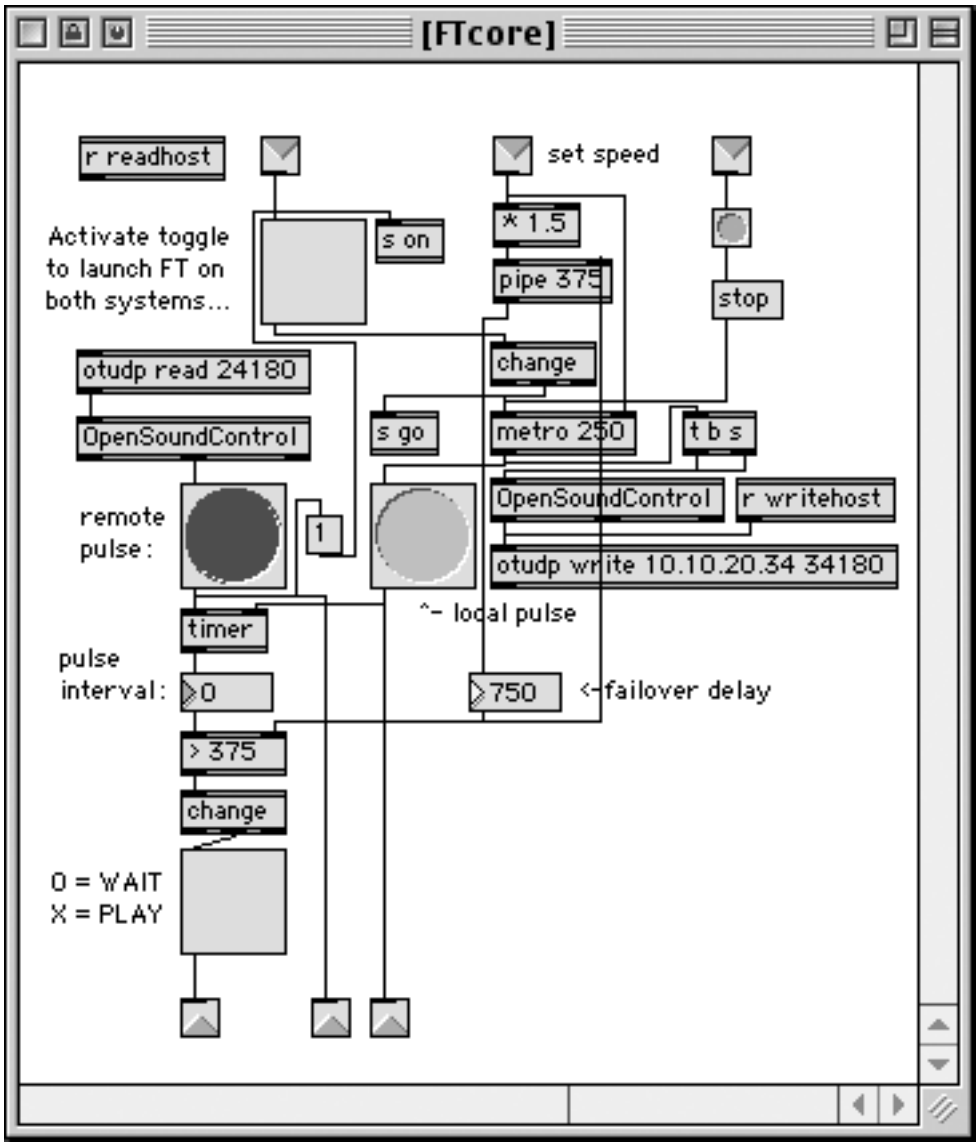
The primary user interface for controlling and monitoring the FT system



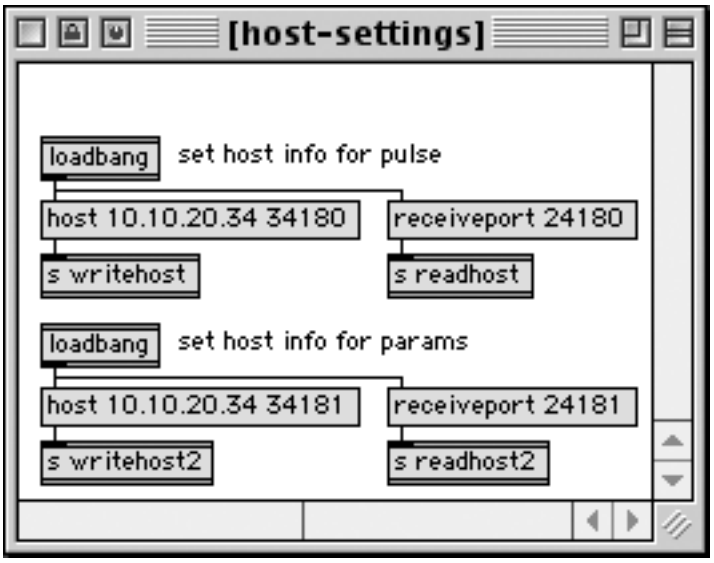
The demonstration interface showing diagnostic information



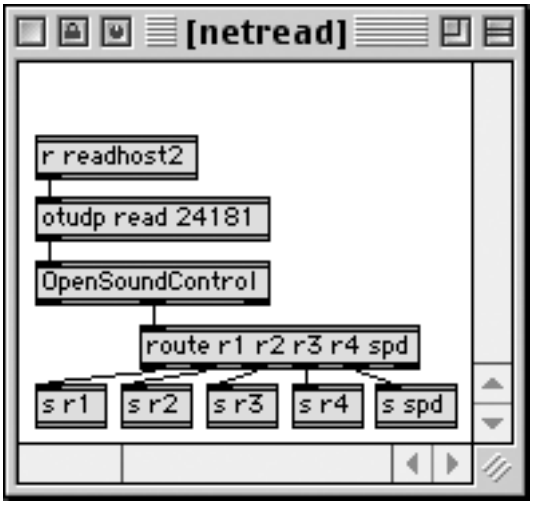
The FT I/O layer that intercepts parameter changes and mirrors them to the standby. If the computer is in standby mode, the gates are closed, preventing activity. If the standby becomes active, the gates are opened, enabling normal processing.



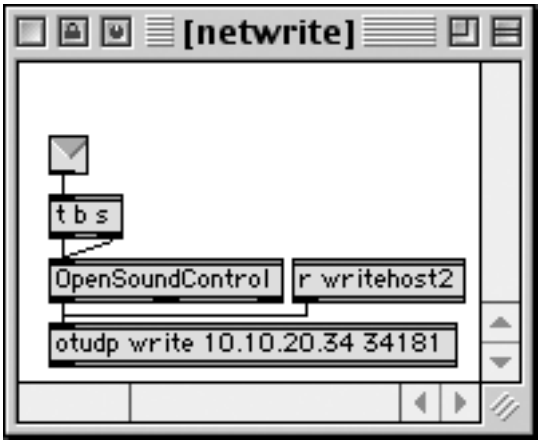
Heartbeat listening, sending, and failover logic (inside FTlayer)



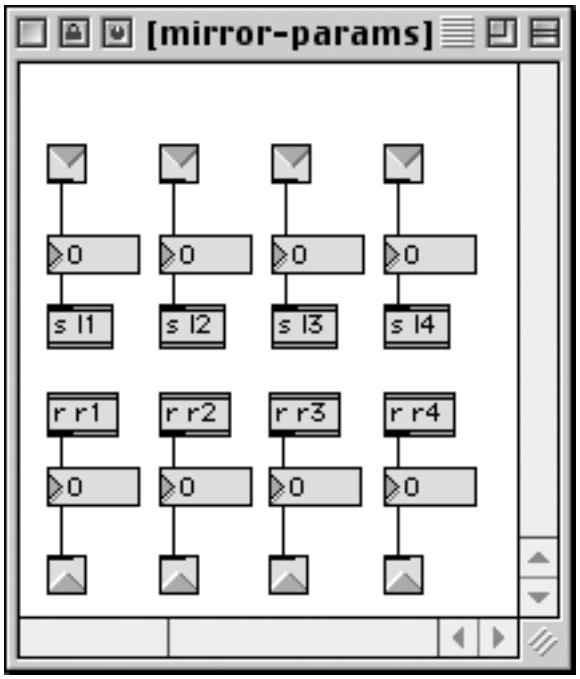
Sets TCP/IP parameters for the otudp external (inside FTlayer). Note the heartbeat and parameters are handled on different ports.



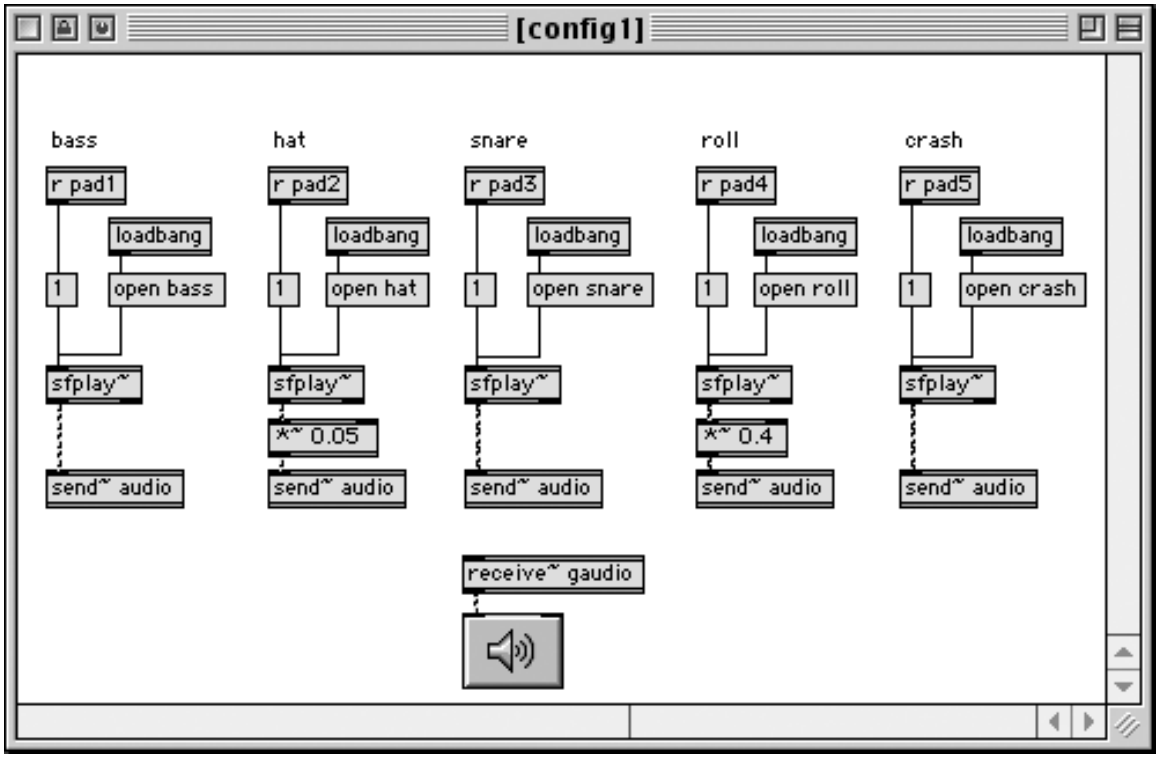
Reads and distributes parameters from the network (inside FTlayer)



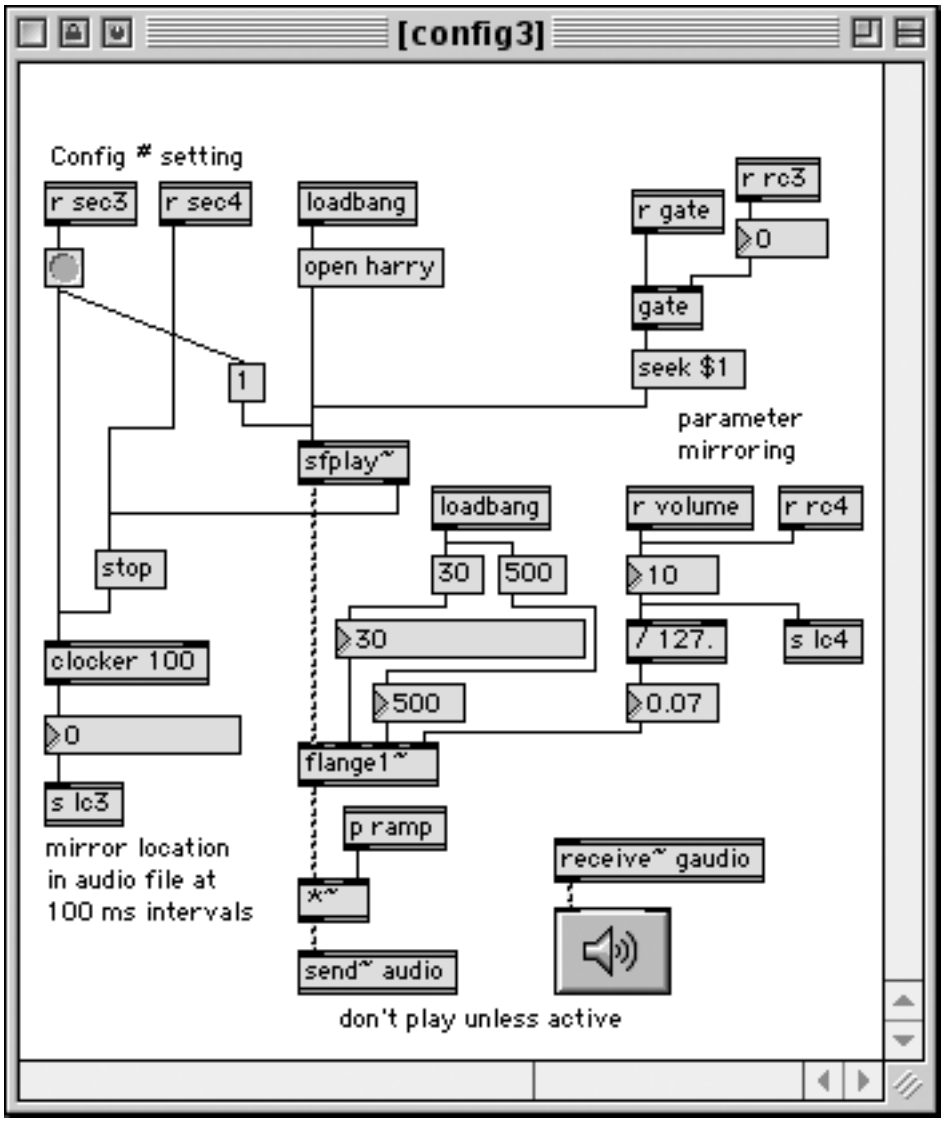
Writes parameters to the network (inside FTlayer)



Mirrors four parameter streams (inside FTlayer).
r1–r4 are remote data incoming from the network.
l1–l4 are local data outgoing to the network.



Triggers samples for Config #1



Manipulates real-time audio filtering for Config #3

And the voice sang on,
piping him back into the dark,
but it was his own darkness,
pulse and blood,
the one where he'd always slept,
behind his eyes and no other's.

— William Gibson, *Neuromancer*