Tempo

A Distributed Media Player and Content Distribution System

Dan Schultz, Martijn Stevenson, and Tom Wilson

6.824 Final Project

Submitted May 12th, 2005

Abstract

Most popular music sharing implementations require powerful centralized servers to keep track of or store the available content. We offer an alternative solution. This paper describes Tempo, a decentralized digital content solution that lets users share their content on the fly as it is being enjoyed. Our system focuses on a community approach to digital media, with multiple content controllers and clients distributing files to other clients while the media plays. The major challenges we faced in this project included efficiently distributing files to an arbitrarily large number of users in a decentralized fashion, allowing multiple users to control media playback, and keeping playback synchronized. We overcome these difficulties by using a BitTorrent-style distribution system, utilizing stateless connections for control messages, and maintaining a synchronized group time for time stamped messages.

1.0 Introduction

Imagine the following. You enjoy listening to music, but you don't have the time or the energy to rip it from your CDs or download it from the Internet. Thankfully, you have an established circle of friends, family or coworkers who share your particular tastes in music. If only there were some way by which your friends could let you listen in. Tempo provides this service.

This paper presents Tempo, a completely distributed music-sharing tool without any central file repositories or predefined controllers. It is a music player, a DJ and a content distributor rolled into one tool. To the user, Tempo looks much like a standard digital music player. The user can form playlists from which he can play songs randomly or in order. In addition, the user can now connect to groups of users formed ad-hoc and listen in on a shared playlist which features music coming from all of the group's controllers. Tempo allows users to create online communities which focus on the enjoyment of shared media.

When creating Tempo, we decided that our system must satisfy the following five properties:

1) It must be completely distributed. The system cannot rely on centralized servers, as these introduce single points of failure.
2) It must support large numbers of clients with relatively high turnover rates. Clients may well be continually disconnecting and reconnecting, so it is important that Tempo handle this gracefully.
3) It must allow distributed control of playback. The whole point of the system is that multiple users can control media playback, so it must not rely on a central controller.
4) It should minimize the delay between a user adding content to the queue and all users obtaining a copy of that content.
5) It should minimize the skew in playback across clients. Ideally, all clients should be in perfect sync with each other.

To enforce these properties, we created a system where all of the clients are identical. While individual Tempo clients act as servers in some respects, there is never any system-wide central control. This enforces properties (1) and (3). To support property (2), we tried to minimize the

amount of state that has to be maintained across clients. By using stateless, transient connections and a randomized gossip protocol to ensure consistency, we were able to significantly reduce the amount of communication overhead and infrastructure required. This makes adding or removing clients a straightforward and efficient process, allowing us to meet our goal. Property (4) is upheld by our file transfer system. We use a modified version of BitTorrent, which allows Tempo to use multiple servers for distributing files without overloading them. This means that files propagate through the system rapidly once the transfer starts. Finally, we enforce property (5) by using timestamps on every message. This allows us to determine when the playback command was executed, which means that we can simply start playing the media at the appropriate moment in the file. We ensure that every client in the system maintains a consistent group time using a time synchronization protocol, described in section 3.4.

The sections that follow illustrate the most important properties of our system. Section 2 describes prior work that influenced the design of Tempo, while section 3 dives into the details of that design. Section 4 discusses interesting implementation details of Tempo, and section 5 presents an evaluation of how well we met our goals. Finally, section 6 considers what future work could be done to extend and improve the Tempo system.

2.0 Related Work

Tempo draws inspiration from a wide variety of sources. One of the important features of Tempo is accessing other clients' media, so related work in file-sharing is a great source of information for us.

The file-sharing aspects of Tempo share qualities with a multitude of applications, including applications such as Kazaa, Gnutella, and FreeNet. These applications let users search for a file they wish to have and allow the user to download the file from other users. Our system is similar, except that we allow the searching and downloading to occur without any user interaction. When a Tempo client needs a file, it automatically sees who has the file and begins downloading it.

The BitTorrent system is also related to the way our file-sharing works. BitTorrent breaks up files into pieces and distributes the pieces individually. When a user wants to download a file, that file could come from any number of users. The pieces are not necessarily sequential and the pieces could all come from one other user or each piece could come from a different user. In order to maximize the speed of file distribution, Tempo is designed to allow file segmentation. We did not include this feature in our prototype, but it could be added with relatively little effort.

The music distribution portion of our system is similar to several streaming and sharing systems in use today. One of the most popular music sharing systems is the iTunes music streaming service, which utilizes Apple's Rendezvous protocol. This system allows users to see the shared music libraries of other users on the local network. Users can play songs from any of the shared music libraries. The main difference is that the user playing a shared song does not get a copy of the song. Also, there is no connection between what the user playing the shared file is listening to and what the user who owns that shared file is listening to. Tempo allows users to listen in on the music of another user.

In addition to the iTunes system, there are several systems available that allow a user to listen to a broadcast of music over a network. A popular network broadcast application is Shoutcast. In Shoutcast, a single controller determines what songs should be played and any number of users can listen to the songs that the controller decides to play. The users listening have no control of the songs being played. In Tempo, any number of users can both listen to and control the music being played on the distributed system.

There are many systems that implement pieces of the functionality desired for the Tempo system. To date, however, none of these systems have incorporated all of the facets of Tempo.

3.0 Design

Our design for Tempo begins with a full-featured media player that has all the bells and whistles of typical media players. We add to this a system for connecting one client to a network of other clients and allowing all of the connected clients to share a common media playlist. Our system facilitates all the necessary file transfers when particular media files are missing from one or more client computers. Tempo also allows for the shared control of the community's playlist and provides the ability for any user to add songs for everyone in the community to hear.

When designing Tempo, we tried to keep our major goals in mind. The overriding principles were that the system must not rely on any centralized servers, and that it must be able to handle large numbers of users who may come and go frequently.

3.1 Offline Operation

When a client is not connected to any other clients over the network, Tempo will run in offline mode. In this mode of operation, Tempo looks and acts much like any other media player. Tempo allows the user to play the files stored locally on the client's hard drive. The typical media player controls are all present: buttons for play, pause, previous file, and next file, sliders for volume and balance, and a progress bar to show the current position in the media being played.

In the offline mode, the user also has access to the necessary input boxes and control buttons to connect to other clients if he so chooses. In this way, the offline media player serves as the entry point for connecting to media communities. The player will seamlessly transition from offline mode into connected operation.

3.2 Connecting

One of the nicest features of a distributed system like Tempo is that new clients can enter the system from any point of access. There are no central servers in Tempo, so clients may connect to any client who is either currently in a group or alone and accepting connections. Every client runs a thread that constantly listens for incoming connections. This allows Tempo communities to grow quickly.

Tempo is flexible about the authentication schemes used when new clients join. One could imagine a range of schemes, from a group password to public-key cryptography. When a new client connects, he will establish a reliable TCP connection to a current client, over which all setup communication takes place. After the new client is accepted into the group and made aware of other clients, this connection is dismantled.

One of the major tradeoffs we had to consider in our design was whether to maintain persistent connections between clients in a community. Currently, Tempo clients do not maintain any open connections. While persistent TCP connections would enhance communication reliability, keeping open network sockets requires a lot of memory and new client overhead. The decision to use unreliable UDP communication allows the system to grow to an arbitrary number of clients. It also minimizes connection establishment overhead, a desirable feature for communities with a high turnover rate. UDP is an unreliable protocol. In particular, using UDP means that packet delivery is not guaranteed, and packet arrival ordering is not guaranteed.

The next two sections will describe how we compensate for the lack of communication reliability.

3.3 Maintaining Group State

The most difficult part of using unreliable communication is that no message is guaranteed to get to its intended recipient. This implies that at any time, any client might be out of sync in some way from the rest of the group. Our solution to this problem is a consistency checker that operates under a gossip protocol. Every connected client runs a consistency thread that contacts random other clients at regular

intervals. Clients send hashes of all shared data structures to other clients. In case the hashes don't match at the receiving end, the other client will send back a sequence of messages representing the most current group state (the client list, the media list, or the last known issued command).

For the system to work decently, we assume that network connections are not intolerably lossy (most messages get through) and that at any point, most clients' states are not wildly divergent. Regardless of network conditions, the system eventually converges to a consistent state because clients are constantly sending each other their latest updates. The system achieves eventual consistency: client states don't always need to be consistent; they just need to be consistent in due time.

3.4 Keeping Things In Order

The second obstacle unreliable communication presents is that messages might arrive out of order. This can be bothersome, especially because all connected Tempo clients need to see (and hear) the same media played in the same order, at the same time. Our partial solution to this problem was that we marked every message (i.e. every event) with a timestamp. We display clients and media as ordered by their creation timestamps. Similarly, we check the timestamp of every media control message (play, pause) to ensure that we only perform the last issued command.

For the timestamp scheme to work, however, we require that the clients' computer clocks all be completely synchronized, or that clients have some notion of group time. We chose the latter. The design tradeoff here was between doing our own time synchronization and using third-party software that would likely set clients' clocks. We felt that it would be more convenient for clients to not have to worry about their computer clock, or to have to change it for that matter. Additionally, maintaining a group time easily lets us do playback synchronization! Our particular solution, then, was to perform a time handshake when a new client connects to the system. Every client stores a time offset from the group time, and thus all clients can perform synchronous playback by skewing incoming timestamps with their time offset and comparing this to local time.

3.5 File Transfer

The primary goal of the Tempo file distribution system is to quickly distribute files to all of the users. Tempo strives to evenly spread the communication load across clients, and to take advantage of local copies of files to accelerate the transfer. To accomplish this goal, we use a BitTorrent-like system where clients contact a known tracker for each file. File transfer in Tempo uses two major subsystems: a Tracker and a File Mover.

3.5.1 Tracker

Each client maintains a database of the files which it has added to the queue. This is called the Tracker, and for every file the client is tracking, it maps the file's unique ID to a list of file segments (chunks) representing this media file. For every chunk, the Tracker stores a server list indicating which clients in the system currently have a complete copy of this particular chunk. These clients can now act as servers for the file chunk. This allows Tempo to spread files throughout the system more efficiently, since every chunk can come from multiple sources. Since every client knows which client added a file to the queue, contacting the Tracker for a particular file is a simple operation. Whenever a client asks the Tracker to provide a server for a chunk, the Tracker checks the requested chunk's server list and responds with the next available server based on its load-balancing policy.

3.5.2 File Mover

Each Tempo client runs a File Mover (FM) subsystem whenever they are connected to a community. The FM subsystem continuously monitors the queue for changes. If a new file is added to the queue, the FM first checks to see if the client already has any chunks of this file on the local disk. It then contacts the Tracker

running on the client which added the file. If any chunks were found locally, the FM informs the Tracker, which will add this client to the appropriate chunk server lists.

For each chunk of the file which is not complete locally, the FM goes through a series of steps. First, it asks the Tracker for an available server. The Tracker returns the server's contact information to the FM, which then contacts the supplied server. When a file request comes in, the server sends the chunk to the client. Once the chunk is complete and stored on the local disk, the FM informs the Tracker that it has the chunk, and proceeds to download the next incomplete chunk.

When designing Tempo, we initially considered having all users retrieve files from a single source (i.e., whichever client added the file to the queue). While this would have greatly simplified our system, we rapidly discarded it on the grounds that a single server would quickly be overloaded in a large system. We also considered building a deterministic distribution tree, with each client communicating with a small number of other clients in a predetermined fashion. While this scheme makes actually transferring the files between users more predictable and efficient by eliminating the necessity of contacting a tracker, it also adds significant complexity. With a tree distribution system, detecting node failures is very difficult. Additionally, in systems with a high turnover rate, a distribution tree would have to be reconfigured often, which causes many headaches. We found that the BitTorrent model, with a single tracker and many servers, was sufficient to meet the goals of our system. Since connections to the Tracker asking for referrals are simple to process, the client running the Tracker will not be overloaded by incoming requests. And since we do not rely on any predetermined distribution model, we do not suffer from the reconfiguration overhead inherent with trees.

3.6 Disconnecting

As with any connected system, clients in Tempo can disconnect in two ways: gracefully (shut down Tempo or press disconnect button) or abruptly (trip over power cord). On a graceful exit, a connected Tempo client sends a message to the group to alert others of his departure. After an abrupt severance from the network, another client will eventually notice a client's absence when the file tracker distributes downloading assignments. At this point, news of the severed client's departure will be publicized.

One decision we made here concerns what we do with media left in the network queue after the client that added it disconnects. In the case where other clients have copies of the data, it might be possible to reassign another client as the content's new owner and tracker. However, consider the case where a client adds some previously unseen media to the shared queue, starts up the Tracker, and abruptly disconnects before any other clients can get a copy. Only the disconnected owner knew whether anyone else had a copy of this data. To deal with risky cases such as this one, we decided to simplify the system and remove any media a client owns from the network queue when that client leaves the group. Conceptually, we feel that this is a cleaner choice since all files in the queue now belong to clients who are active.

After a graceful disconnect, a Tempo client seamlessly transitions into offline operation, as described in section 3.1. From there, the client can use Tempo once again as a local media player or to connect to a different Tempo community.

4.0 Implementation

4.1 Overview

We constructed a prototype of the Tempo system to attempt to meet all of the goals outlined above in the introduction. We implemented the prototype in approximately 3500 lines of Java code. We chose to implement our system in Java instead of other languages

such as C++ for several reasons. The most important reason is that Java's motto of "write once, run anywhere" allows us to run our prototype on Windows, Apple OS, or *nix based systems. We also feel that ease of development gives an edge to Java because of its simple handling of multimedia through the Apple QuickTime libraries for Java.

We decided to use the Java libraries for the Apple QuickTime media player for handling all media in Tempo. QuickTime is versatile as it is able to handle many standard formats for media types from music to images to movies. The alternative would be to use the audio libraries that come standard with Java. However, these libraries do not natively support the MP3 file format, which is problematic for a media application.

In order to stay flexible while still meeting our design goals, we tried to use good modular design principles. This way, when we find improvements that can increase performance of the system, it should be simple to make these improvements without affecting a large portion of the underlying classes. For instance, if we find a new algorithm for tracking files, we can simply rewrite the Tracker, maintaining the same interface and ensuring that other classes will be unaffected.

Finally, we used multithreading along with the necessary locking to improve performance. Multi-threaded subsystems allow us to perform various tasks concurrently and also prevent one subsystem from delaying the methods running in other subsystems.

4.2 File Distribution

The implementation of our file distribution system bears particular note. In order to simplify the code for our prototype, we did not feel it was necessary to implement file chunking. In other words, we use entire media files as the chunks. However, this does not significantly impact the performance of the Tempo system, as shown below in 5.0.

To keep multiple connections (both incoming and outgoing) from overloading Tempo's distribution system, we used a series of multi-threaded objects. The File Mover system maintains one thread for monitoring the queue, another thread for downloading files, another listening for referrals from trackers, and one thread per referral connection. This allows Tempo to get a great deal of concurrency out of its FM system, greatly enhancing the system's performance.

The Tracker system is implemented with a HashTable data structure mapping file IDs to chunk server lists. Currently, the load-balancing policy used by the Tracker is a simple round robin scheduling process. As the Tracker refers clients to servers, it places the server at the back of the server list. While this algorithm may not be the most desirable, it ensures that servers are seeing roughly equal numbers of referrals, which is sufficient to prevent any one server from experiencing an overload situation.

5.0 Evaluation/Performance

We will now demonstrate that we achieved with our implementation of Tempo the five properties set forth in our introduction. Our test bed for all performance data were four Pentium IV PCs of varying speeds, three of which were linked by a 100 Mb/s Ethernet, and one of which used a 11 Mb/s wireless network. We will now go through our goals one by one and examine the extent to which we met them.

1) *Our system must be completely distributed.*

Indeed, the system is completely distributed. The only central points of failure are trackers, clients who add media to the network queue and monitor its distribution. In case of tracker failure, we remove said media from the network queue, and the system continues to function.

2) *It must support large numbers of clients with relatively high turnover rates.*

While we had no more than four machines on which to test our implementation, we can make predictions about performance for larger client groups. The majority of upkeep traffic related to client turnover comes in bursts of UDP messages. Clients publicize the arrival and removal of members and media with UDP messages sent to all other clients. Because these messages are stateless, they require little overhead. Our tests demonstrated that a Tempo client could create and send 100,000 UDP messages in an average of 4.183 seconds. Even with ten thousand users in a group, this sort of output rate should not adversely affect performance.

3) *It must allow distributed control of playback. The whole point of the system is that multiple users can control media playback, so it must not rely on a central controller.*

This is solved in a similar manner to 2). Any playback controller simply sends out a command message to all clients, which we showed to be feasible even for large client groups.

4) *It should minimize the delay between a user adding content to the queue and all users obtaining a copy of that content.*

This was the focus of most of our performance testing. The message delays and file transfer overhead together represent how long a connecting user has to wait before the first song starts playing. Keep in mind that as playback continues, any content acquisition rate greater than the data playback rate will results in uninterrupted playback.

First we measured how quickly our consistency checker propagated information across the client base. We purposely did not send out a message, waiting to see how long it was before *every one* of the four test clients had received the message via random consistency hops. We varied the consistency-checking interval between 50 ms and 400 ms, noting reasonable results for all these configurations. With our initial 100 ms setting, it took an average of 1.690 seconds for the consistency messages to travel two to three hops from the source. Note that this is an ultimate worst-case scenario that occurs when only one UDP message gets to its intended recipient. The log-scaled graph shows that we can expect delays to increase logarithmically as message intervals increase. Because the spread of consistency information grows exponentially, we can also expect the time delay to increase logarithmically as more clients enter the system.

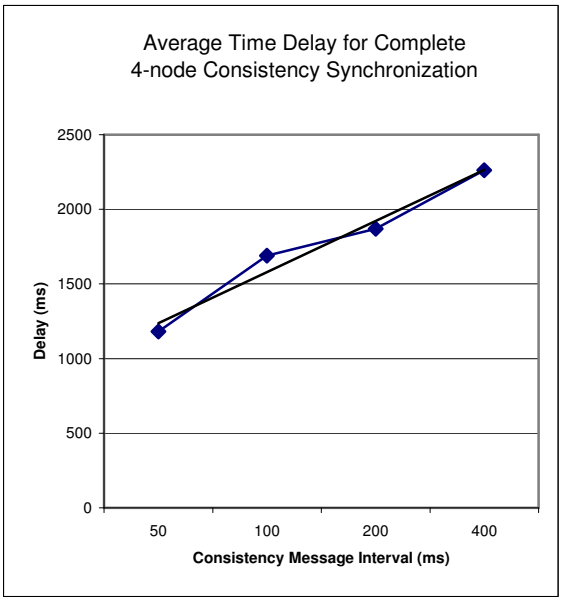

Figure 1: This graph shows on a log-scale how long it took on average for consistency checkers to propagate information from one source client to three other Tempo clients.

We did several tests to determine how long it takes for a data file to spread itself across a small Tempo community. We measured the time from when the user pushes the "Add Song" button to when the last user informs the Tracker that the song has been downloaded successfully. We call this time the latency. We used three different setups for this trial. C1 had four computers: three desktops and one wireless laptop. C2 had consisted of only the three desktops, and C3 was C2 plus one more desktop. We conducted two types of test for each community. First, we tested the latency with a song that all of the computers had local copies of. Second, we tested latency with a file that none of the computers (except for the source)

Table 1: File Propagation Latency Results

Community	File Location	Latency (ms)	Size (Mb)	Average Speed (Mb/sec)
C1	All local	1234	NA	NA
C1	None local	12,172	5.21	.428
C2	All local	768	NA	NA
C2	None local	2547	2.52	.989
C2	None local	2375	3.42	1.44
C3	All local	847	NA	NA
C3	None local	3047	5.70	1.87
C3	None local	22,079	77.3	3.501

had local copies of. The results are summarized in Table 1.

Clearly, C1 had the lowest performance. This is easily explained by the presence of a computer using a wireless (11 Mb/sec) connection, compared to the wired (100 Mb/sec) connections on the desktops. Even so, Tempo performed quite adequately. Assuming the standard conversion of approximately 1 Mb per minute of music, Tempo was distributing at least 30 seconds of music every second. This performance is perfectly acceptable.

Admittedly, these tests were performed over MIT's network, which is incredibly fast. However, it is plain to see that Tempo is not adding significant overhead to the file transfer process. This means that Tempo should be able to perform adequately even when the underlying network is not particularly fast.

5) *It should minimize the skew in playback across clients. Ideally, all clients should be in perfect sync with each other.*

As mentioned in 3.4, we use time handshakes to tell all clients the group state. This means that as far as the computers know, playback is perfectly synchronized. We humans did qualitative testing, listening to playback from multiple computers to discover time skew. More often than not, we found that Tempo clients synchronized well enough for comfortable listening, also across computers with wildly varying clock times.

6.0 Future Work

While implementing our prototype, we came up with several ideas for improvements we would like to make and features that we would like to add to Tempo. The first improvements fall in the category of user interface refinements. For instance, we would like to include a progress bar to keep the user informed of files' download status in the network queue. Another example of an interface refinement would be to allow users to select which panes they wish to see. For instance, some users may not be interested in which users are connected to the community, while others do not care for the equalizer.

Another improvement we would like to make is in the Tracker. In our current implementation, files are distributed in one big chunk. We believe that as user communities grow, a better balance of file distribution can be maintained if files are broken into smaller chunks for independent distribution. This change would only affect the Tracker classes and would use the same interface as the one chunk file Tracker.

Because the current Tempo implementation uses the QuickTime libraries to handle media and because QuickTime handles so many file types, we began to consider the possibility of playing movie files in our system. If we added a pane for graphical output, users could watch movies together just as they play music together in the current implementation.

Finally, we would like to add the ability for users to search for pre-existing communities. Perhaps with the use of web server queries, we

could allow users to search for communities that fit their musical tastes, their geographic location, their age group, or any number of other criteria. This way, when someone downloads the Tempo client, he will not need to know others in the community from the start. He can simply search for open communities and connect to any user in the community of his choosing.

7.0 Acknowledgements

The designers and developers of Tempo would like to thank Robert Morris and Athicha Muthitacharoen for their design advice and guidance during 6.824. The class rocked.