COMP 50CP Final Project Report

Composte

Make bad music together

Team

Export-By-Email

- Tom Magerlein
- Robert Goodfellow
- Wesley Wei

What is Composte?

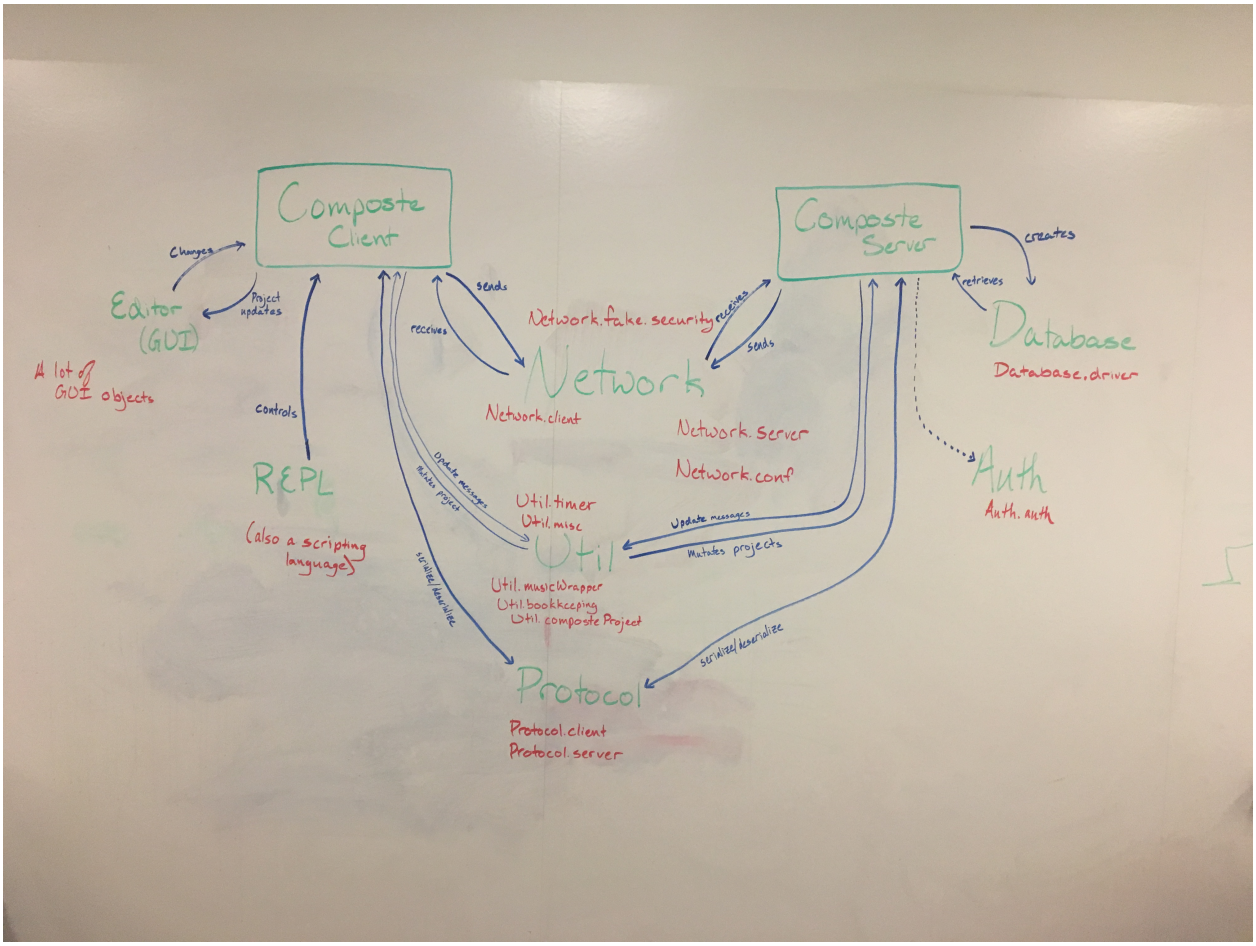
Composte is an application that allows collaborators to concurrently collaborate on music composition. Think of it as Google Docs for music, but bad.

This submission

This submission can do/supports the following:

- Multiple concurrent users working on projects
- Display notes
- Edit notes
- Change MIDI instrument
- MIDI instrument change
- Accidentals
- Automatic Treble Clef
- Set Tempo
- Project-local chat/text-to-speech
- Impoverished scripting language
- More impoverished scripting language

Design/Architecture



Composte builds a simple RPC to separate servers from clients. This RPC is built on top of several modules, described below.

auth

Authentication of username/password pairs.

database

Encapsulation of database interactions.

protocol

Encapsulates serialization and deserialization of messages.

util

A collection of miscellaneous utilities, including music handling and a generic REPL.

network

Encapsulates communicating over a network.

The music processing backend processes music.

util

The music processing backend is implemented entirely with music21, a music processing engine developed at MIT. A thin wrapper around the music manipulation functions was necessary in order to perform the RPC calls intelligently. Playback is indirectly handled by music21, which delegates to pygame, which attempts to delegate to a combination of timidity and freepats.

The GUI

The primary developer has declined to comment.

The REPLs

Because the goalposts were moved so many times over the course of the project, it became clear to us that we would not be able to get the GUI-driven interaction patterns that we wanted. Therefore, we fell back to text-driven interaction patterns. To this end we thought that it would be prudent to create a couple of REPLs from which to drive client interactions with servers.

Outcome

Somehow, it appears to work.

Alternatively, we have two impoverished REPLs driving a GUI and interactions with a remote server. We have moved the goalposts enough so that this is indeed the minimum deliverable.

</sarcasm>

Looking back at our design

Our design at its core has three somewhat disjoint tasks: Maintaining and mutating an internal representation of music, GUI-driven collection of changes to music (Actually REPL-driven), and remote procedure call (RPC).

While we still believe that this model is fairly elegant and practical, we see many flaws in our implementations of the modules backing these tasks.

The most prominent lesson we learned was that GUI development is difficult and time-consuming. While there are plenty of powerful, well-documented GUI toolkits out there for Python, we didn't find anything aimed at providing a sheet music editor. In this sense we failed to foresee how much of a pain developing the GUI would be, given that none of us have experience developing a GUI of this nature. It turns out that making an interactive graphical editor is hard.

Therefore, we fell back to driving the client primarily with a terminal-based REPL. This reduced the amount of GUI work that was required for us to both test and use our application. While text driven interaction is the antithesis of a graphical music editor, we believe that it is still ultimately useful, as a scriptable interface is always a nice feature. To be clear, we believe a full blown GUI is important, but we also believe that we have neither the time nor the expertise to pull one off in the 7 hours before the demo.

The internal representation of music, is sadly not cleanly separated from the GUI. In order to slightly reduce the amount and magnitude of headaches Tom had, is having, and will continue to have, the internal representation of music is somewhat tailored to the graphical features that we would like the GUI to have.

When attempting to integrate the GUI with the music backend, several deficiencies in the design of the backed were discovered. Firstly, because rests have no direct representation (they are the absence of notes at some point in the score), the GUI has no point of reference to allow it draw rests. Furthermore, there is no backend concept of measures whatsoever; they are a fiction imposed by the GUI. This made Tom sad. This also created the need for an entire extra layer of abstraction within the GUI's structure. Lastly, the playback engine can only play one stream at a time, and because music21 streams are composable (as in function composition), if a score were modelled as a stream of streams rather than a list of streams, it would be possible to play back all parts at once and it would make more sense semantically.

The way communication over the network is implemented under our RPC is somewhat unsatisfying. Because we chose to collect all incoming client messages into a single REQ/REP ØMQ socket, we do not have the notion of separate connections. This hurts us in two ways: only one message can be processed at a time on the server, and broadcasts go out to *all* connected clients, not just the ones subscribed to the project of interest. A better underlying network model would have the concept of separate connections, as that would allow both concurrent processing of unrelated messages and the restriction of broadcasts to the clients that it is intended for. A migration away from the REQ/REP network model is warranted, but discovered late enough to prove challenging to change without pushing us past the deadline.

This means that currently, most concurrency issues in this application do not have to do with the server processing messages from clients. This will hopefully change in future releases.

In the face of how glaringly *bad* the network model is, we have found the actual network endpoints to be surprisingly pleasant to use.

Division of Labor

It turns out that we're terrible at fair division of labor, but that we're marginally better at dividing the problem into subtasks. We focused on our own tasks and then came together to compose an application out of the parts we built.

Tom Magerlein:

- GUI
- Debug console

Robert Goodfellow:

- Internal representation of music
- Diff Handling

Wesley Wei:

- RPC
- Deployment/Official Server
- Generic REPL

Unfortunately, we did not dedicate sufficient time to documentation to produce documentation of acceptable quality. Much of the documentation we have is out of date and no longer correct.

Bug/Spontaneous Feature Tracker

- Backend ties don't work in playback

    - Ties, though not represented on the GUI, can still exist in the internal representation. Unfortunately they are not respected when performing playback of music.
    - We have determined that this is a bug underneath the `music21` package that we use to do playback. We have submitted a ticket.
    - See this issue
    - The bug manifested itself when playback did not perform as expected. We found it by observing that playback was inccorrect. We noticed that playback was incorrect when we attempted to playback music containing a tie. If we had tried to playback music containing ties earlier, we would have found the bug earlier.

Source Code Tourist's Guide

For your viewing pleasure or displeasure, we provide an elided `tree` of the repository.

```
Composte
├── auth
│   └── auth.py
├── client
│   └── < GUI Suffering >
├── ComposteClient.py
├── ComposteServer.py
├── data
│   ├── composte.db
│   └── users
│       └── < User data >
├── database
│   └── driver.py
├── demoScripts
│   └── < Demonstration scripts >
├── doc
│   ├── architecture
│   │   └── index.md
│   ├── deliverables
│   │   └── index.md
│   ├── images
│   │   └── Bad_Diagrams
│   │       └── < "Diagrams" >
│   ├── index.md
│   └── protocol
│       └── index.md
├── Dockerfile
├── html
│   ├── index.html
│   └── styles
│       └── composte.css
├── logs
│   └── < Logs >
├── network
│   ├── base
│   │   ├── exceptions.py
│   │   ├── handler.py
│   │   └── loggable.py
│   ├── client.py
│   ├── conf
│   │   ├── logging.conf
│   │   └── logging.py
│   ├── dns.py
│   ├── fake
│   │   └── security.py
│   └── server.py
├── protocol
```

```
│   ├── base
│   │   └── exceptions.py
│   ├── client.py
│   └── server.py
├── README.md
├── requirements.txt
└── util
    ├── bookkeeping.py
    ├── classExceptions.py
    ├── compositeProject.py
    ├── misc.py
    ├── musicFuns.py
    ├── musicWrapper.py
    ├── repl.py
    └── timer.py
```

Source Descriptions

Top Level

`CompositeServer.py` implements a complete Composite server on top of the network server.

`CompositeClient.py` implements most of a Composite client on top of the network client.

auth

`auth.py` contains functions to create and verify password hashes.

database

`driver.py` encapsulates access to the the database. It translates between database schemas and python objects.

network

`client.py` provides a network client.

`server.py` provides a network server.

`dns.py` provides methods to get ip addresses from domain names.

network/base

`exceptions.py` contains exceptions the network clients and servers expect users to raise in the event of trouble.

`handler.py` provides a base class for a stateful message handler. Users may choose to derive their message handlers from this.

`loggable.py` provides a base class to provide simpler logging.

network/fake

`security.py` provides classes conforming to the `encryption_scheme` interface that the network servers and clients expect, but provide no encryption.

network/conf

`logging.py` contains the default logging configuration that the network clients and servers use, as well as a method to read that configuration.

client

This module contains the GUI.

doc

This directory contains documentation of the project and deliverables for COMP 50CP. Much of the documentation is out of date, and a fair amount may also be unreachable from internal links.

protocol

`client.py` contains methods for serializing and deserializing client messages.

`server.py` contains methods for serializing and deserializing server messages.

protocol/base

`exceptions.py` contains exceptions that the `protocol` module may raise.

util

The `util` module implements a number of miscellaneous things that are needed to make the server work. Notably, the music backend is implemented here

`bookkeeping.py` implements two static object pools.

`classExceptions.py` provides some exceptions used in the class hierarchy used by the GUI.

`compositeProject.py` provides the internal, in-memory representation of a project. This also provides serialization and deserialization facilities.

`misc.py` provides a function to get the version (commit) hash.

`musicFuns.py` provides the mutators for the internal representation of music.

`musicWrapper.py` provides a thin wrapper around `musicFuns.py`, conforming to the message handler contracts that `CompositeServer` expects.

`timer.py` provides a method to run a function at a configurably approximate interval.

`repl.py` provides the REPL shell/scripting interface that Composite has decomposed into. The REPL is likely to have access to all methods that the GUI exposes to the user, and indeed may have finer-grained access to music manipulation methods. The REPL is feature-impoverished, but capable of just enough to be semi-useful.