Audio Engineering Society

Convention Paper 10183

Presented at the 146th Convention
2019 March 20 – 23, Dublin, Ireland

Turning the DAW Inside Out

Charles Holbrow[1]

[1]*Massachusetts Institute of Technology*

Correspondence should be addressed to Charles Holbrow (charles@media.mit.edu)

ABSTRACT

Turning the DAW Inside Out describes a speculative, internet-enabled sound recording and music production technology. The internet changed music authorship, ownership, and distribution. We expect connected digital technologies to continue to affect the processes by which music is created and consumed. Our goal is to explore an optimistic future wherein musicians, audio engineers, software developers, and music fans all benefit from an open ecosystem of connected digital services. In the process, we review a range of existing tools for internet enabled audio and audio production, and consider how they can grow to support a new generation of music creation technology.

1 Introduction

Recorded music had a huge impact on the twentieth century, with analog and digital recording technologies co-evolving along with changes in musical tastes and possibilities. The evolution of sound recording technologies will continue in the coming century. How will the processes of making and recording music change? We are beginning to see how cloud computing, streaming media, machine learning, and digital social networks are changing the music landscape.

Audio technology creators today face a difficult problem: learn how to make connected, intelligent technology, or risk falling behind competitors that understand the opportunities better. We do not claim that the technology described below is the only way to proceed. However, it does reveal unexplored possibilities, and expose pitfalls that creators of modern connected audio software are already falling for. It is our hope that sharing these ideas will enable the audio engineering community to collectively think beyond naive initial assumptions, buzzwords, and current trends, and help to advance audio engineering in the twenty-first century.

We begin with a comparison to a tool for software development: git. Git is both an open source software project, and a protocol. It makes new kinds of software engineering practices possible. Commercial products built on the git foundation are offered through organizations like GitHub, GitLab, and Bitbucket. Services like GitHub do not only make collaboration easier, they also host third party services (for example, automated testing and continuous integration), and serve as a portfolio and resume for software engineers. This is the kind of model we envision for next generation audio tools. However, music is not the same as code. We can learn from the architecture of git, but the modules that we need to build next-generation audio engineering tools are different, if not more complicated.

Our goal is not to design "GitHub for DAWs," but to imagine how an ecosystem of open and connected services can enable new possibilities for musicians and for music. What are the modules and protocols that will be

needed for music software in the 21st century? Before answering this question, we will walk through a hypothetical (science fiction) recording session, describing how it is different from a conventional recording session today.

2 The Science Fiction Studio

A band's producer and bass player, Jordan, opens up a session in her DAW of choice to review pull requests. A few days ago, the band was jamming in the studio, and found a groove that Jordan wants to make the basis of a track. After the studio session, she sent a link to the session to a few friends, asking for feedback. The track received a few comments, and a pull request with scratch vocals from a topliner. She reviews the comments, taking note of which ones to share with the band later, and merges the pull request for the rest of the band to review.

Jordan also subscribes to a song arranging service. After the band finished recording, the automated service analyzed the full three hour jam session, and identified three distinct songs that the band played. The service created three new sessions in Jordan's account, with markers for intro, verse, and chorus. The tempo map generated by Jordan's native DAW was reasonably accurate, but the commercial service handles time signature changes better, and accurately re-organized an hour's worth of material into a three and a half minute arrangement. The service also added symbolic transcriptions of the performances: A transcription of the chords, and MIDI-like note transcriptions for the individual tracks.

Jordan plugs in her bass guitar, hits the play button in her DAW, and records a few overdubs to comp a bass track that replaces the scratch track recorded in last week's jam session with a polished version over the three minute arrangement. When she presses 'play' in the DAW, the audio from her bass is recorded and cached on her local machine, but also sent to the cloud were it can be processed, archived and analyzed. When she is satisfied, she can forward the result to the rest of her band for more overdubs. An automated service can send quotes for overdubbing a professional string ensemble in a high-end studio. When the band is ready to release the song, it can be fingerprinted, along with the rights and authorship. From there, artists and bots can sample and remix the final track, all while retaining references to rights and attribution information.

Jordan can give multiple services read access to her stems, and write access to a directory for mixdowns. These might be bots that do automated mixes, individual mix engineers, or combinations of the two. Jordan can then give read access to the mix down directory to a streaming service, which will compensate her, her band, and the mixing and mastering engineers if the song becomes popular. The streaming service annotates the master mixes with playback statistics. Other services can read the statistics, and these can drive further mixes, or advise Jordan on how to structure her band's next single.

At any point during this process, Jordan can publish a rough mix of a new track to streaming services. The mix might not be very polished, but it will only be available to the band's immediate supporters. If Jordan allows it, supporters who hear the initial mixes can make pull requests, fork the project, or request permission to sample it.

We can imagine many services could fit into a system like this. Chord progression and arrangement recommendation services. Educational music theory or arrangement aids. Services that automatically repackage stems as sample libraries and sell licenses, or save samples to the performer's private collection. Bots that scrape the network for illegally copied material. Many components that could fit into a system like this are on the market today. Splice[1] lets users backup, share and fork DAW sessions. iZotope Neutron[2] will generate channel strip presets that attempt to automatically mitigate mixing artifacts like masking. Hooktheory[3] visualizes and compares chord progressions from a database of thousands of songs. Propellerhead[4] and others offer a centralized "app store" that sells or rents audio effects, sample libraries, and synthesizers. Ohm Studio[5] and Soundtrap[6] are real time collaborative cloud based DAWs. These services leverage the current internet platformization trend, where success is measured by market share. Following these services to their logical conclusion results in a convergence in the tools used to create music, and a convergence of the music itself.

[1] https://splice.com/
[2] https://www.izotope.com/en/products/mix/neutron.html
[3] https://www.hooktheory.com/
[4] https://www.propellerheads.se/
[5] http://www.ohmstudio.com/
[6] https://www.soundtrap.com/

An alternative future is one that takes inspiration from the success of git. In this future, many developers can build modules on top of an open source protocol without getting locked in to a platform. Following this pattern to its logical conclusion reveals a diverse ecosystem of tools for different kinds of collaboration and music creation. What follows is a proposal for the technical foundation that such a system could be built on.

3 Terminology

In the technical description below, the following terms should be considered explicit:

- **Annotations** — Time stamped information or instructions about a session, track, or asset.
- **Artists** — The user, who is creating music.
- **Audio Asset** — The raw digital encoding of a wave form.
- **Metadata** — Data about tracks and assets. For example, "composer", "performer" "conductor".
- **Plugin** — An extension hosted by a DAW.
- **Resource Access Interface (RAI)** — The software interface for services to access an artist's sessions and assets.
- **Service** — Services are the modular tools for processing and annotating audio sessions. Services can accept and return audio, audio sessions, symbolic audio, or metadata. As an example, a service might accept an audio session as an input, and return a timestamped list of chord changes (annotations). Another service might accept a song session as input, and provide streaming access to a mastered version that song for fans to listen to.
- **Session** — The file saved by a DAW that includes arrangement, routing, plugin, assets, asset references.
- **Symbolic Audio Asset** — The raw music asset, encoded symbolically. For example, a MIDI file.
- **Track** — A rendered session, often a stereo audio file.

4 Technical Requirements

How could a system like the one described above work? The goal in this paper is not to design each of the modular building blocks like automated musical analysis, transcription, and mixing. Instead the aim is to describe the system that these modules could be built on top of (see figure 1). Ideally such a system would provide music creators the freedom to utilize a variety of diverse tools in their creation process, while incentivizing researchers and developers to create modular tools and plugins that can work together in creative and interesting ways.

Traditionally, the music production workflow follows a sequence (for example: composition, recording, production, distribution). When all of these operations are performed in a cloud enabled toolchain, we may stop thinking of them as a sequence: A track's composition and production can continue even after the track has been released to streaming services. That track can be sampled, covered, or remixed by other artists and collaborators. Every step can be automated, and every creative and technical decision can be data driven. All these capabilities can be integrated into the toolchain.

How can next generation composition and recording tools take advantage of collaborative and data driven processes, while simultaneously fostering creativity and diversity in the results? This new approach to creating and recording music represents a paradigm shift from the conventional approach, and the supporting tools will need to evolve proportionally.

Our inner designer might be tempted to imagine the user experience of working with the DAW of the future. With the experience in mind, we can then work backwards to the technical implementation. This approach is inadvisable, because it forces us to answer many unanswered design questions simultaneously. It is too early to envision and design the full experience. We cannot create GitHub without first creating git.

Instead of trying to re-imagine the entire music pipeline, let us start with the foundational blocks that we can build new musical tools on top of. Essentially, this involves turning the DAW inside out and exposing its contents to the cloud, where artists and services can sample, remix, and interact. If we do this right, new creative workflows will emerge naturally. We can evaluate how artists are using the tools, and use what we learn to inform further development.

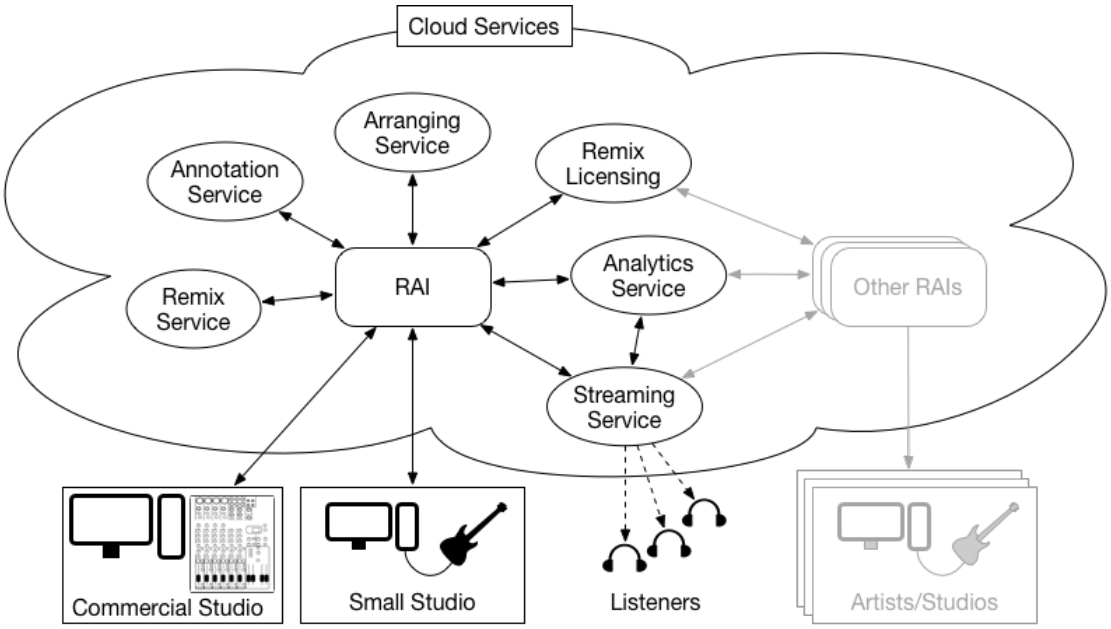

Fig. 1: The structure of the RAI and surrounding services, showing how the contents of a DAW session are exposed to the web. Note that authentication and authorization controls are present, but not shown.

5 Comparison With Git Objects

Returning to our earlier comparison, the git version control system is based on four simple binary objects: Blob, Tree, Commit, and Tag [1]. A git repository uses the host's filesystem to store these objects, named by their sha1 hash. These primitive git objects and underlying git procedures make a complex variety of commands and collaborative workflows possible. The basic git commands are responsible for creating new objects, and updating local and remote file systems based on the existing objects. The commit object encapsulates attribution, and that attribution travels with the object when it is pushed or pulled to and from a remote repository. Additionally, git does not attempt to solve dependency and package management, leaving that to other compatible or complimentary services like pip[7] and npm.[8]

What would a system look like for music? Git objects were designed to describe state in the evolution of a codebase, and to facilitate branching, merging, and merge conflict resolution. The way that code is created is very different from the way the music is composed. The way we share musical ideas is very different from the way we share software. Consequently, we should not expect the same primitive objects to be the optimal choice for working with music.

Instead of mimicking the capabilities of git (version control, branching, and merging), and applying those to music, this is a design proposal for foundational infrastructure that could enable the kinds of collaborative and computational music creation processes described in section 2. With that in mind, consider the following description of musical asset 'objects' that could integrate into the music creation process.

6 Music Objects

The role of a music object is to create and manage addressable relationships between assets, annotations, and metadata. For example, consider a stereo mixdown of a drum kit. There are many different ways it could be used. It can be a scratch track for overdubs. It can be deconstructed as a Rex[9] sample. It can be transcribed, and converted to a symbolic asset. It can be used as the basis for a grove template to be applied to another track.

[7] https://pypi.org/project/pip/
[8] https://npmjs.com
[9] https://www.propellerheads.com/recycle

It can be distributed as part of a sample library. In any of these forms it can serve as a data point for generative music or music information retrieval. If libraries of music assets are saved and searchable through their annotations and metadata, we can build automated services that store and retrieve the stereo drum sample for any of the use cases described above.

All information pertaining to an asset will be accessible through a content addressable system. Like git, resources will be identified by their 40 character hexadecimal sha1 hash. This allows them to be saved in a filesystem like the git objects in a local repository, or unambiguously referenced by a URI.

7 Musical Metadata and Annotation Serialization

The document that represents a user within a service like Facebook will include a collection of properties and relationships typically wrapped in HTML or JSON. For conventional web services like Facebook, the schematic that defines properties and relationships that make up the user profile data is not indented to be used externally. Every conventional web service that implements user accounts is responsible for defining and maintaining an internal 'user' schematic.

It would be possible to design cloud based music production tools using this conventional approach. Each service could design its own data models, and be responsible for parsing the annotations of upstream services. A chord annotation service would implement its own data structure modeling a song timeline. That service would access a client's audio assets, save the chord annotations in a proprietary format, and return the result to the client. This is essentially the model that exists today, and is employed by services like Splice.com. This approach is not without advantages. Primarily, it is simple. Each service developer can define the schematics that are most suitable to their needs. Because a service developer will write their own schematics, they will have no trouble understanding it. This approach benefits from the "separation of concerns" design principle. Separate service developers do not need to understand any one particular resource definition language, and may use whatever method they are most familiar with.

However, our goal is to facilitate many services that interact with each other. The existing platform based model facilitates the ecosystem that we already have:

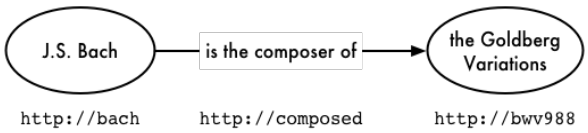

Fig. 2: An example RDF "triple."

Each platform measures success by its market share. Each benefits from the "network effect" only when its users are exclusive.

The Web Ontology Language (abbreviated as OWL) and Resource Definition Framework (RDF) created by the W3C provide a standardized language for explicitly defining ontological schematics and relationships, and describing data within those schematics. The W3C languages are designed with exactly this use case in mind: The ontology and knowledge graph are not owned or managed by any one particular service. Instead, ontological primitives like resource classes and relationships can be distributed across a many different web servers, and can be referenced with a URI. As an example, consider the following triple (shown in fig. 2):

> J.S. Bach is the composer of the Goldberg Variations.

The RDF would divide this knowledge into three parts: A subject (J.S. Bach), a predicate (Is the composer of), and an object (the Goldberg Variations). Using RDF, each of the three parts could be referenced by a URI. That way, many web resources can access the same predicate, assigning an axiomatic and machine readable relationship between people and compositions. The semantic web is designed such that the ontological structure can be defined in a distributed fashion using the existing DNS and HTTP protocols.

Music information retrieval researchers proposed a standardized way to describe audio features, including annotations and metadata. Fazekas and Sandler [2] describe how a software application such as a DAW could read and write metadata in a standardized but extensible way. Their proposal takes advantage of the RDF, and the standards proposed by the W3C for semantic "linked data" on the web. An ontological standard provides a major benefit to DAW, service, and plugin developers. Fazekas' proposed ontology standardizes metadata descriptors like "artist name" and

"composer," as well as musical timeline features like "beat location" and "chord change." This allows many different services to recognize each other's annotations and contributions. Cannam et al. [3] describes how the RDF provides a convenient mechanism for developers to create additional annotation types, and publish these specifications to the web.

In the examples above, many different services perform operations on an artist's assets, annotations, and metadata. If schematics are written in an explicit well defined format, developers can organize their services around shared schematics, eliminating redundancy, and improving interoperability. If the musical annotations supported by the existing music ontology are not sufficient, services can also design, document, and publish new annotations that update or extend the existing ontologies. This means that schematics must be general, and developers must learn to work with the RDF.

Consider the example from the beginning of this document. The artist's session and assets were exposed to several different services. Each service added and updated assets of its own. All services need to be able to parse each other's metadata and annotations. At the same time, each service may define its own extensions to the existing schematic.

8 Resource Access Interface

In the example at the beginning of this paper, Jordan allowed a third party service to access and update her DAW session. Another service was given read access to her session and tracks, and provided her with additional annotations. How can she control access to her session and track assets? All her assets need to be accessible on the internet. The server that makes assets accessible needs to authenticate and authorize read and write access requests from services and individuals.

8.1 Uploading Assets

A practical implementation of this service could work in the following way. First, assets created during a recording must be uploaded to the resource server. A software daemon on the recording engineer's computer monitors the DAW's asset directory, similar to services like Dropbox and Google Drive. When a change is detected, any newly created or updated assets are uploaded to an asset server. This can be accomplished with a one way file synchronization algorithm as is done by the rsync software utility [4].

The process of recording music generates a significant amount of metadata, including but not limited to the time and place of the performance, the musicians involved, recording techniques used, all of which can be semantically linked to the musical composition and arrangement. A discussion of the semantic properties and workflow for authoring metadata resulting from a recording session is described in detail by the multitrack ontology [5]. Recording techniques such as microphone type and placement may also be described by the studio ontology[10]. Ideally, all available metadata on a recording session would be added at the time of the recording session, and could be edited and updated by the recording engineer as well as by the artists. This brings us to an additional technical requirement of the resource server: It must expose an interface to create, read, update, and delete assets, annotations, and metadata.

8.2 CRUD Operations

There are two options for performing create, read, update, and destroy (CRUD) operations on linked data. The first is via SPARQL queries. SPARQL is a query language made specially for linked data, and standardized by the W3C. It has a very flexible query syntax, that allows complex queries on linked data. The specialized query language does come with a cost: Because queries written in SPARQL are designed with the flexibility to traverse a linked data graph, a poorly written query can overwhelm server resources. This can be mitigated by using a intermediary like D2RQ[11], which translates SPARQL queries to SQL for operating on a traditional relational database which has shoehorned in linked data.

The second option is to use standard HTTP methods and headers, which is similar to a traditional REST HTTP interface. This method is proposed in the Socially Linked Data (SOLID) specification.[12] The SOLID REST specification includes some additional requirements, for example, support for wildcard "globs"

[10] http://isophonics.net/content/studio-ontology

[11] http://d2rq.org/d2r-server

[12] https://github.com/solid/solid-spec/blob/61b88d9c2022039896bca910c05b25da056c1cae/api-rest.md

when retrieving data with the HTTP GET method. The SOLID project also includes a declarative specification for group based authentication and authorization called Web Access Control.[13] Developing a REST style interface is made easier by the many existing implementations, and well established middleware based approach for authenticating and authorizing requests. However, it cannot support the more complex queries provided by SPARQL explicitly made for searching and manipulating linked data.

8.3 Metadata Storage

Compared to uncompressed audio assets, annotations and metadata consume a small amount of storage. The simplest option for saving linked data on the server is to use the server's filesystem. In this model, metadata would be saved as linked data files, in the .rdf or .n3 formats supported by the W3C. For example, a .n3 file that encodes musical annotations using the Timeline and Music ontologies [6] may be referenced by:

https://example.com/m/46eee28edc90[...]/timeline.n3

Where a hexadecimal string (following /m/) is the sha1 hash of the audio asset that the timeline refers to. Using this pattern allows simple services to run with minimal configuration on an Apache or Nginx server.

However, modern web services require more advanced authentication, authorization, data validation, backup, queries, and scale than a filesystem based approach provides. For this reason, the conventional approach is to put a HTTP server in front of a database, which allows more flexibility and customization. A full discussion of issues surrounding linked data and databases is out of scope. We will highlight a few key factors.

If the entire linked dataset is small enough to fit on a single server, a centralized datastore may be used. For example, Jena2 [7] is a Java based toolkit for storing and querying subject-predicate-object triples in a centralized SQL database. Jena2 supports input and output in RDF serialization formats including N3 and RDF/XML. It also supports SPARQL queries. If the linked dataset cannot fit in a single machine, a distributed RDF store must be used. D-SPARQ (Mutharaju et al. [8]) is an RDF query engine built on MongoDB, a NoSQL datastore designed to scale vertically and horizontally across many servers. D-SPARQ uses the Map/Reduce functionality built into MongoDB, but does not accept SPARQL queries. Another scalable option is provided by Amazon Web Services. AWS Neptune[14] is a managed distributed graph database with support for RDF objects and SPARQL. An up-to-date discussion of many additional RDF ready database options can be found in Sakr et al. [9].

[13] https://www.w3.org/wiki/WebAccessControl

8.4 Storing Assets

Audio assets are larger than metadata, and should be stored and exposed through a different mechanism. In the proposed model, services can query annotations and metadata. If a service needs to query the audio itself, it can request the audio directly. Artists should have control over which services may access which assets, metadata, and annotations.

A simple way to store large binary files like audio assets on a web server is using that server's file system. However, this is often avoided because it offers limited support for best practices like versioning and automated, redundant, distributed backups. A more flexible open source option would be GridFS, an official specification and API for storing large files in MongoDB collections. GridFS takes advantage of the built in replication and backup procedures made for MongoDB. Alternatively, commercial services can provide turnkey solutions for saving large binary blobs, and exposing those blobs to the world wide web. AWS S3 allows web clients to upload and download large assets on behalf of a service. Access to assets stored in AWS S3 buckets can be subject to customizable authentication and authorization parameters.

9 DAW Sessions

We have described how a cloud enabled service could act as a repository for assets, annotations, and metadata. What about DAW session files? The RDF coupled with the music ontology provide a powerful collection of standards for metadata and annotations. However, the services described in the introduction don't just edit metadata; they integrate into the digital audio production workflow. One of the proposed services autonomously parsed Jordan's DAW session, automatically organized the jam into an intro, verse, and chorus,

[14] https://aws.amazon.com/blogs/aws/amazon-neptune-a-fully-managed-graph-database-service/

and saved a derivative version. To do this, services must be able to access and de-serialize the session information. Access is not complicated. Session files can be uploaded and distributed through the cloud in a similar fashion to audio assets. However, a service that modifies a DAW session must be able to parse the session file. There are many very different session file formats, and interoperability between the formats supported by different DAWs is limited.

9.1 Parsing DAW Sessions

One option is creating a new DAW from scratch with the cloud service based workflow in mind, and publishing the session spec. This allows us to build support directly into the DAW, simplifying the implementation of complex features like real-time collaboration. This ensures compatibility, but means we need to create a DAW, and motivate artists to use it. This might seem inadvisable, because DAWs are a very competitive space, but many new platform-based collaborative DAWs are in are currently in development (examples include Soundtrap, Ohm Studio, BandLab, and Amped Studio). However, our goal is not to create a new platform in the conventional sense, so we will be better served supporting existing audio production tools.

Another option is authoring a session format specification. An "edit decision list ontology" in an RDF compatible format is one potential format. This would allow us to create session converters that (for example) convert the .RPP session format saved by Reaper to our portable format. We could also ask DAW developers to include support for exporting to our custom format.

Alternatively we could use an existing session file specification. There is historical precedent for open DAW standards. In 1993, Avid Technologies published the Open Media Framework, which was designed to allow interoperability between different DAWs [10]. Two subsequent standards AES31 [11] and AAF [12] published in 2000 and 2004 respectively, where created for the same purpose. Unfortunately, true DAW interoperability is non-trivial. The AAF format includes support for a sample accurate edit decision list, fades, and to some extent, automation. However, it does not support routing, bussing, MIDI, and audio plugin configuration. Creating a standard that exactly reproduces every possible parameter in a DAW is impractical, because of the many evolving capabilities of different audio tools. Existing commercial DAW session translators like AATranslator[15] and Vordio[16] suggest that explicit limited support for certain features is a more realistic approach.

AAF encoding is supported by Studio One, Pro Tools, Logic, Nuendo, Digital Performer, Final Cut Pro, and Premier, but not by Reaper, Ableton Live, FL Studio, Bitwig Studio, Ardour or Reason. Could we expect cloud services to adopt the AAF format? There are some obstacles. Unlike the RDF standards published by the W3C, there is no process in place for extending or maintaining the specification. Also, unlike the RDF, legacy formats like AAF do not include any way to publish extensions to a given specification in a distributed fashion. The defining feature of the RDF is support for a continuously evolving extensible definitions. Only a C++ library exists for parsing AAF files. Meanwhile, modern and generic binary formats like Protocol Buffers[17] have libraries in many different programming languages, as well as built-in support for versioning and backwards compatibility.

Standardizing DAW session file format involves obstacles whether we write our own specification or use a legacy format. These challenges can be circumvented by leaving the problem of parsing session files entirely to service developers. To create the session restructuring service that organized Jordan's session file into song sections, each service could advertise exactly which session file formats it supports. This would fragment service interoperability, but also encourage DAW developers to expose their session file specifications to service developers.

A final option involves trading the service capabilities for operational simplicity. We can limit service interaction to raw audio assets. In this scenario, an artist interested in tapping a service would render audio files from their session, before uploading them for processing by a cloud service. This significantly limits the type of available services. For example, services could not programmatically alter a mix, and could not automatically publish those mixes to customer facing streaming services.

[15] http://www.aatranslator.com.au/

[16] http://vordio.net/

[17] https://developers.google.com/protocol-buffers/

9.2 Branching and Merging DAW Sessions

Some of the example services described in sections 2 and 6 create new assets (for example, sample libraries, stereo mixes, analytics, or annotations) from existing assets. Another category of service facilitates collaboration between musicians and services, mirroring collaborative aspects of GitHub. Musical collaboration services (for example, a services that enable pull requests between divergent DAW sessions) have a non-trivial technical requirement: a mechanism to handle branches and merges with support for merge conflict resolution.

Branching and merging is an important part of distributed collaboration, but today, no service or DAW supports merge conflict resolution. Synchronization services like Dropbox and Splice.com allow users to revert to a previous version, or to fork a session, but no DAW or platform as implemented merging.

While merge conflict resolution for DAWs is an unsolved research question, existing research on the algebra of patches used by modern version control systems for code [13, 14] is a starting point. The algebra of patches (used in version control systems like git) is typically based on difference and merge algorithms for raw text (such as GNU diffutils [15]). Meanwhile, a DAW session file may be encoded as semi-structured data. For example, a hierarchy of audio tracks in a session file can be represented with a tree data structure, and encoded with XML as opposed to raw text. For resolving merge conflicts between two divergent sessions with a known common ancestor, three-way difference and merge algorithms are required. A 2015 publication by Autexier [16] presents appropriate algorithms with the explicit goal of allowing those algorithms to be tuned for the application domain.

10 Summary

We described how music composition, recording, and production could evolve given a cloud-enabled redesign with a distributed architecture and a focus on open and extensible standards. One way to visualize this is "turning the DAW inside out", and exposing its contents to a spectrum of human and automated collaborators, with the goal of encouraging new and previously unimaginable music production processes. This is in contrast to conventional DAWs that work by embedding assets and plugins, centralizing the workflow, and (often) by locking users in to a particular platform.

We described the technical direction for how we can start working toward this platform, using existing standards when possible. Some implementations details are left to future work. A robust authorization interface is required for artists to specify which services may access which assets. An mechanism for resolving merge conflicts between divergent branches is a requirement for true distributed collaboration. We discussed bundling metadata with assets with the implication that when an artists samples or remixes existing tracks, attribution can be built in to the production process. This is a important step, but we still need a payment or transaction system that enables upstream artists to be compensated.

The comprehensive vision considers how creating music can continue to develop in an age when we no longer need to clear boundaries between music composition, recording, production, and consumption.

References

[1] Hamano, J. C., "GIT — A Stupid Content Tracker," in *Linux Symposium*, volume 1, pp. 386–394, Ottawa, Ontario Canada, 2006.

[2] Fazekas, G. and Sandler, M., "Ontology-Based Information Management in Music Production," in *Audio Engineering Society Convention 126*, 2009.

[3] Cannam, C., Jewell, M. O., Rhodes, C., Sandler, M., and D'Inverno, M., "Linked Data And You: Bringing music research software into the Semantic Web," *Journal of New Music Research*, 39(4), pp. 313–325, 2010.

[4] Tridgell, A., "Efficient Algorithms for Sorting and Synchronization," *Doktorarbeit Australian National University*, 1999.

[5] Fazekas, G., Raimond, Y., and Sandler, M., "A Framework for Producing Rich Musical Metadata in Creative Music Production," in *Audio Engineering Society Convention 125*, 2008.

[6] Raimond, Y., Abdallah, S., Sandler, M., and Giasson, F., "The Music Ontology," *ISMIR 2007: 8th International Conference on Music Information Retrieval*, 8, pp. 417–422, 2007.

[7] Wilkinson, K., Sayers, C., Kuno, H., and Reynolds, D., "Efficient RDF storage and retrieval in Jena2," *Proceedings 1st International Workshop on Semantic Web and Databases*, pp. 35–43, 2003, doi:citeulike-article-id:926609.

[8] Mutharaju, R., Sakr, S., Sala, A., and Hitzler, P., "D-SPARQ: Distributed, scalable and efficient RDF query engine," *CEUR Workshop Proceedings*, 1035, pp. 261–264, 2013, ISSN 16130073.

[9] Sakr, S., Wylot, M., Mutharaju, R., Phuoc, D. L., and Fundulaki, I., *Linked Data: Storing, Querying, and Reasoning*, Springer International Publishing, 2018, ISBN 9783319735153.

[10] Lamaa, F., "Open Media Framework Interchange," in *Audio Engineering Society Conference: UK 8th Conference: Digital Audio Interchange (DAI)*, 1993.

[11] Yonge, M., "AES31 Audio File Interchange," in *Audio Engineering Society Conference: UK 15th Conference: Moving Audio, Pro-Audio Networking and Transfer*, 2000.

[12] Tudor, P., "The Advanced Authoring Format and Its Relevance to the Exchange of Audio Editing Decisions," in *Audio Engineering Society Conference: 25th International Conference: Metadata for Audio*, 2004.

[13] Lynagh, I., "An Algebra of Patches," 2006.

[14] Baudis, P., "Current Concepts in Version Control Systems," *CoRR*, abs/1405.3, 2014.

[15] MacKenzie, D., Eggert, P., and Stallman, R., *Comparing and Merging Files with Gnu Diff and Patch*, Network Theory, 2003, ISBN 9780954161750.

[16] Autexier, S., "Similarity-Based Diff, Three-Way Diff and Merge." *International Journal of Software & Informatics*, 9(2), pp. 259–277, 2015.