Distributed time-centric APIs with CLAPI

Paul Weaver and **David Honour**
Concert Audio Technologies Ltd.
Reading, UK
{paul, david}@concertdaw.co.uk

Abstract

Distributed control of applications by multiple simultaneous devices has traditionally been achieved via protocols such as MIDI or OSC. These simple protocols require additional semantics, often communicated out of band, in order to construct meaningful APIs.

We present the Concert Light-weight API (CLAPI) framework: a session-based pub/sub API framework that aims to simplify the definition and usage of semantic, time-centric distributed controls.

Keywords

API, Distributed, Pub/Sub, Semantics, Introspection.

1 Introduction

The Concert Light-weight API framework (CLAPI) is one component of our larger distributed DAW project. It grew from our need to control an audio engine from a heterogeneous mix of clients simultaneously, with event-driven feedback of the evolving state of the system.

1.1 Open Sound Control

Our original efforts sought to build semantics on top of Open Sound Control (OSC) [Wright, 2002]. We had hoped to use OSC to communicate instructions and state across the network. However, we ran into some issues with that approach:

- TCP/session-oriented support was lacklustre. This caused problems, for example, when considering TLS/authentication.
- Establishing bidirectional communication was hard (only OSC servers can receive messages), which would complicate NAT traversal.
- The semantics around variable length lists of values weren't standardised (if they were even present at all).
- Only OSC bundles could be timestamped (c.f. individual messages) and bundles could be nested. This meant we couldn't always derive timing information when required.
- Bundle nesting also meant we had trouble building sensible error semantics.
- The dispatch rules provided by existing libraries weren't particularly dynamic.

In other words, whilst OSC is a perfectly good protocol, it did not fit our pattern of component communication as well as we'd have hoped. This led us to consider the problem domain more broadly, and start experimenting with our own API protocol/framework.

2 Network API Paradigms

Before we consider our specific API requirements, we will briefly discuss three approaches for controlling remote systems. We consider both user-facing controllers, such as hardware devices or software GUIs, and autonomous systems connected to the network.

2.1 Fire and Forget

Conceptually, unidirectional protocols like OSC and MIDI [MIDI manufacturers association, 1996] are very simple. Once a connection has been established, control data is transmitted from the client (the party triggering the operation) to the server (the party doing the work) when an action is desired. For example, MIDI sends explicit instructional events like "Note On", which can be fairly directly translated into method calls on an instrument or captured by a recording device for later playback.

OSC is similarly intended to be used in an instruction-centric way, albeit that it is more agnostic in its design (no specific instructions/methods are defined in the protocol itself). Its human-readable metadata also offer a sub-

stantially more direct representation of semantic intent.

The unidirectionality of these protocols has some implications. They do not, for instance, provide a mechanism for applications to report errors. This is assumed to be noticed "out of band", frequently by the user. This "fire and forget" mentality implies, in the absence of such feedback semantics, a controller/executor relationship between the components of the system.

There have been attempts to create feedback semantics for these protocols [Portner, 2017], but they are far from universally supported.

Unidirectionality has the additional consequence that every receiver must handle (even if only by discarding) the union of all possible messages, due to the sender being unable to determine the recipient's capabilities.

2.2 Remote Procedure Calls and Request/Response

The most common kind of network API paradigm arranges the exchanges between client and server like that of a local function call: a request is made by the client to named method ("endpoint") on the server with some arguments and a response is received synchronously after the action is completed.

The most widespread example of this request/response pattern is HTTP [IETF, 1999], and many remote procedure call (RPC) API frameworks are based on top of it [Winer, 1999; W3C, 2000]. However, HTTP itself is an RPC protocol in its own right, with a fixed set of actions (HTTP methods such as GET, POST etc.) with their own arguments (headers), semantics and responses (exit code and potential body).

Feedback to method is improved over unidirectional communication as the client does not have to assume that the invocation was successful. This means that state can be kept in sync without any out-of-band communication, at the expense of more complicated client side handling.

The Representational State Transfer (ReST) philosophy [Fielding, 2000], of which HTTP is an embodiment, attempts to limit the proliferation of ad-hoc methods by structuring requests to the server in terms of *resources*, with a fixed set of methods providing predictable behaviours on those resources.

Formalisms like JSON schema [Andrews/IETF, 2017], and HAL [Kelly, 2011], aim to aid discoverability and introspection by building on top of ReST principles.

2.3 Publish/Subscribe

Another form of API that has been gaining traction, particularly in distributed systems, is the publication/subscription (*pub/sub*) model.

In APIs of this type clients (*subscribers*) communicate to providers (*publishers*) which data they wish to be informed about. Any subsequent updates about the data are then sent to the subscribers who have chosen to be notified.

Most commonly, pub/sub APIs are very scalable message queuing systems [ISO/IEC, 2014; RabbitMq, 2018; MQTT, 1999; OASIS, 2015; Hintjens et al., 2014a; Hintjens et al., 2014b], and clients connect to an API broker, rather than to the source of the data directly.

3 CLAPI's Paradigm

We introduced each of the above network API paradigms because CLAPI has a mixture of features from all of them.

At its heart, CLAPI is an idempotent pub/sub API framework. Providers publish state updates to an API broker (the "Relay") and interested clients subscribe to subsets of that state to receive updates.

Unlike traditional message queues, the Relay keeps a local cache of the application state in memory, so that subscribers are notified of the *current state* of data when they subscribe as well as any future updates.

CLAPI, however, is not just a broadcast system. Just as in traditional "fire and forget" systems, clients can push state update messages of their own, and the Relay forwards them to the provider of an API. Responses to these messages are not received synchronously, as in regular RPC, but rather through existing subscriptions.

These state update semantics give us a nice mix of properties for building an event-driven distributed application. Furthermore, CLAPI incorporates discoverability, introspectability and validation into the API framework from the ground up.

In the next sections we detail the mechanics of CLAPI and continue to contrast it with the three prevailing paradigms we have covered above.

4 Data model

The data communicated by CLAPI are conceptually held in the leaf nodes of a tree and are addressed by paths of names, such as

`/api/version`. The container nodes can also be addressed, e.g. `/api`, and can be thought of as containing data about the names and ordering of their children.

Each top level path (e.g. `/api`) is handled as an isolated API namespace and is "owned" by a single client, who is referred to as the *provider* of that API. The provider may not subscribe to their own API, but may subscribe to other APIs over the same connection. Other clients cannot directly modify the provider's API, but can publish update messages which are validated and forwarded to the provider only for handling.

Before a provider can publish any data, it must provide a collection of types that fully specify the form of the data at every path. Unlike most other API frameworks, this schema is also event-driven and updates can be published at any time. This allows providers, for instance, to expose only session-loading controls until a session is selected, or to defer providing type information for a plugin until after it has loaded.

4.1 Types of Time

There are two notions of time, often not explicitly distinguished, in session-based audio applications: *wall-clock time* and *project time*. CLAPI distinguishes between them explicitly.

Wall-clock time is the time we experience—the one shown by most clocks and watches. It is monotonically increasing and cannot be stopped.

Project time is the time between the start of the recorded work and an event occurring. It is mapped to wall-clock time by playback. This is useful for talking about the relative positions of events that will occur during playback (for parameter automation, for instance).

The values at the leaves of a CLAPI tree can change over project time, or they may be fixed. If the data may change, we refer to the node as a *time series* of *time points*. Time points consist of a pair of time value and tuple of data values, and are indexed in the series by UUID so that we limit the impact of messages crossing on the wire.

Times are stored in an NTP-inspired manner as a pair of 64- and 32-bit unsigned integers representing seconds past the Unix epoch and the sub-second fraction respectively.

The structure of a CLAPI tree is fixed over project time. Changes to both tree structure and project-time data can be made at any point in wall-clock time, and are always applied immediately.

Name	Constraints
enum	Option names (required)
time	
word32	Bounds
word64	Bounds
int32	Bounds
int64	Bounds
string	Regular expression
ref	Type name (required)
list	Item schema (required)
set	Item schema (required)
ordSet	Item schema (required)
maybe	Item schema (required)

Table 1: Value schema types

4.2 Schema

Leaf nodes in CLAPI are referred to as *tuples*, and consist of either a single heterogeneously typed tuple of values, or a time series thereof if the value is to change over project time.

Container nodes can either be *structs* (with a fixed set of heterogeneously-typed children) or *arrays* (with a variable set of homogeneously-typed children).

Because each of these entities has different constraints, there are three kinds of type definition in CLAPI, as detailed below.

4.2.1 Tuples

The type definition for a tuple consists of a documentation string, an ordered mapping of field names to value schema and an *interpolation limit*.

All documentation in CLAPI is intended for human consumption when exploring an API, and has no semantic meaning within the framework.

Each value schema consists of the type of value accompanied by any constraints on that type. For example, it is possible to specify that a value can be any 32-bit integer, or a list of strings that conform to a particular regular expression. The supported value types and their constraint options are shown in table 1. Note that container schema like *list* are defined recursively by constraining with an item schema that is itself another entry from the table.

CLAPI can express interpolation between the time-series data points in each tuple tree node. This means that applications do not have to

send dense streams of data to produce smoothly varying control values.

If the values in a tuple node can change over time, each tuple of values in the project-time series is associated with interpolation parameters. The permitted interpolations are:

Constant This tuple will remain as specified until the next time point.

Linear This tuple is linearly interpolated to the next time point.

Bezier This tuple is interpolated via a Bezier spline (parameters supplied by the user) to the next time point.

The interpolation limit, defined in the tuple type definition, specifies what kinds of interpolation parameters can be specified for each tuple. If the values in the tuple will not change over project time, the interpolation limit is specified as *uninterpolated*. Otherwise, because each of the above kinds of interpolation is more expressive than those that precede it, the interpolation limit simply takes the form of the most expressive interpolation type allowed for the tuple.

CLAPI does not attempt to restrict the choice of interpolation limit according to value types—it is perfectly possible for a provider to publish an API that states it can do Bezier interpolation on strings, and it's the provider's job to do whatever would be expected of it in that situation.

4.2.2 Arrays

The type definition for an array consists of a documentation string, and a type name and permission information about the children of the array. The type name specifies that any direct child nodes of this container node will be of the named type. We call the permission information the *liberty* of the child nodes. It is selected from the following enumeration:

Cannot The client cannot supply this data. Should the client create a new array element containing a path with this liberty, the provider will generate a value for it.

May Paths with this liberty are editable. Should the client create a new array element containing a path with this liberty without supplying a value the provider will generate a default value.

Must Paths with this liberty are editable. Should the client create a new array element containing a path with this liberty they must supply a value.

4.2.3 Structs

The type definition for a struct consists of a documentation string and an ordered mapping of child names to pairs of type name and liberty. Structs in the tree must always contain all their defined children. The liberty value, however, allows for partial definition of struct data by clients when inserting structs into array containers, which providers must then fill in. In other words, defining liberty values on structs allows us to nest structured data within arrays whilst keeping the semantics around defaults and read-only behaviour.

4.3 Attribution

Situational awareness is important in an application with collaborative control. That is, we want to know not only what changes have been made, but by whom. CLAPI attaches an *attributee* to each piece of data and each child in arrays, in order to keep track of who is doing what in the session.

4.4 Introspection

Because providers must publish a collection of types that fully specify the type of every path in their tree of data, and because the Relay publishes type information about the root node that contains all the providers' API namespaces, it is possible to explore the entire CLAPI data space beginning with a single subscription to the root node.

This means that CLAPI APIs are both discoverable and self-documenting, with a limited and consistent set of semantics—desirable properties we detailed in our brief discussion of ReST (section 2.2).

Type assignment messages are sent to clients when they first subscribe to a path to prevent them from having to infer the type of a path by traversing down from the root node type.

4.5 Consistency

Data updates received by clients must always lead to a self-consistent tree state. For example, tuples must contain data of the correct type, and data cannot be assigned to paths that are not reported to be contained in a parent node.

Therefore, multiple changes may be communicated together and applied atomically. This

is similar to bundles in OSC. Because of the dynamism of our type system, it is often required that type changes are accompanied by corresponding data changes.

The kinds of operations that can be performed in each set of changes differs with respect to client role and communication direction, due to the restrictions laid out in section 4. The kinds of information that can be transmitted between each party are outlined in table 2.

Given these consistency restrictions, and our general data type constraints, we include semantics for error reporting in our message exchange. Error message strings are keyed in relation to the API entity to which they pertain. We call this key the *error index* and it can take one of the following forms:

Global The error is not specific to any particular piece of data (e.g. an error decoding a message).

Type The error is specific to a type (e.g. referencing a type name that does not exist).

Path The error is specific to a path (e.g. attempting to assign invalid project-time-global data, or changing the child keys of a struct).

TimePoint Indices of this type contain the path and UUID for the point to which the error pertains (e.g. attempting to assign invalid data to a specific point in a time series).

5 Other concerns

5.1 Time

Sometimes it is important for a client to know when an event occurred even if that client was not connected when that event happened. CLAPI messages are timestamped to high precision so that the Relay may present its own API with information about the time differences between clients.

5.2 Topology in Larger Deployments

API providers can subscribe to other APIs within the same Relay, or even make connections to other Relays in order to collect information about remote systems that they may then choose to expose. This allows the formation of substantially more complex topologies withut the requirement for consensus algorithms in CLAPI.

6 Ecosystem

Our current implementation of CLAPI is written in Haskell. We have written library code that implements building blocks required to write a CLAPI application [Concert Audio Technologies, 2018b], including types for values, definitions and messages, as well as serialisation. We have implemented the Relay application using the library.

We have produced a dummy API provider in Haskell for testing purposes. The audio engine component of our application is currently written in a mixture of C and Haskell, with the Haskell portion providing the high-level API interaction and control plane.

We are looking to provide a framework for creating HTML5/WebSocket interactive frontends for CLAPI applications. These take the role of clients in the solution. This component is in the early stages of development at the time of writing [Concert Audio Technologies, 2018a].

We hope that the high degree of type introspection possible with CLAPI can assist in creating a UI by allowing the dynamic generation of widgets for controls. This should mean that clients and providers do not need always to be kept in tight version synchronisation. We aim to blend this dynamism with some explicit layout design in order to provide useful, customisable interfaces.

7 Future

We are currently prototyping our distributed DAW on top of the CLAPI framework. The design of CLAPI is heavily influenced by what we are trying to achieve in our application and vice versa. As both the application and CLAPI are still under very active development, we appreciate that some details may change between the time of writing and the conference.

We are curious as to whether the mixed-paradigm approach and features like validation, discoverability and introspection, which we have tried to incorporate in the CLAPI framework, are applicable to a wider range of applications outside our problem domain. We'd also like to explore further how these features impact the design of applications, and whether there are any technical considerations we may have overlooked in CLAPI's design.

Ultimately, we hope that CLAPI will be of use to the community, either directly, or by stimulating discussion about the kind of high-level features we want in our APIs in the future.

	Definitions	Type Assignments	Data Updates	Errors
Relay → *Client*	•	•	•	•
Client → *Relay*			•	
Relay → *Provider*			•	•
Provider → *Relay*	•		•	•

Table 2: Information each role can communicate to others in CLAPI

References

H. Andrews/IETF. 2017. Json schema specification. http://json-schema.org/specification.html.

Concert Audio Technologies. 2018a. A Prototypical CLAPI web GUI. https://github.com/foolswood/elmweb.

Concert Audio Technologies. 2018b. Clapi. https://github.com/concert/clapi.

Roy Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine.

Pieter Hintjens et al. 2014a. Zeromq distributed messaging. http://zeromq.org/.

Pieter Hintjens et al. 2014b. Zeromq message transport protocol. https://rfc.zeromq.org/spec:23/ZMTP.

IETF. 1999. Hypertext Transfer Protocol 1.1 (RFC 2616). https://tools.ietf.org/html/rfc2616.

ISO/IEC. 2014. ISO/IEC 19464 - Advanced Message Queuing Protocol (AMQP) v1.0 Specificiation. https://www.iso.org/standard/64955.html.

Mike Kelly. 2011. Hypertext application language specification. http://stateless.co/hal_specification.html.

MIDI manufacturers association. 1996. MIDI 1.0 standard. https://www.midi.org/specifications/item/the-midi-1-0-specification/.

MQTT. 1999. MQTT homepage. http://mqtt.org/.

OASIS. 2015. Mqtt version 3.1.1 plus errata 01. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

Hanspeter Portner. 2017. OSC additional semantics. https://open-music-kontrollers.ch/osc/about/.

RabbitMq. 2018. Rabbitmq amqp implementation homepage. https://www.rabbitmq.com/.

W3C. 2000. Simple object access protocol 1.1 specification. https://www.w3.org/TR/soap/.

Dave Winer. 1999. Xml-rpc specification. http://xmlrpc.scripting.com/spec.html.

Matt Wright. 2002. Open sound control 1.0 specification. http://opensoundcontrol.org/spec-1_0.