Transparent Communication Within Multiplicities

Angelo Fraietta, Oliver Bown
UNSW Art and Design
Sydney, Australia
angelo@smartcontroller.com.au, o.bown@unsw.edu.au

Sam Ferguson
University of Technology Sydney
Sydney, Australia
Samuel.Ferguson@uts.edu.au

***Abstract*—The Internet of Musical Things is an emerging field of research that intersects the Internet of Things, human-computer interaction, ubiquitous music, artificial intelligence, gaming, virtual reality and participatory art through device multiplicity. This paper introduces a paradigm whereby data points and variable parameters can be strategically mapped or bound using aliases, data types and scoping as an alternative to flat address-structured mapping. The ability to send and/or access complex data types as complete entities rather than lists of parameters promotes data abstraction and encapsulation, allowing greater flexibility through modular architecture as underlying data structures can change during the lifestyle or evolution of a computer based composition. Additionally, the facility to define data accessibility, and the ability to reuse human readable names based on a variable's scope is a common feature of most programming languages. This paradigm has been extended in that scoping a variable can be dynamically bound or addressed to specific objects, class types, devices or globally on an entire network. We describe the evolution of this paradigm through its development via various project requirements.**

I. Introduction

The Internet of Musical Things (IoMusT) is an emerging field of research that intersects the Internet of Things (IoT), human-computer interaction, ubiquitous music, artificial intelligence, gaming, virtual reality and participatory art through device multiplicity [1]. Artists are creating spatial and portable media experiences that exploit the capability afforded by smart devices to interact and coordinate with one other [2]. In addition to devices sending sensor information to one another in the IoT domain, networks become scalable in that devices can be added or removed from them [3]. As a result, composers are changing the way they deal with works that contain multiple devices. In the same way that a graphic designer might consider drawing to a display as a single entity rather than a set of pixels, composers using multiplicities are choosing to treat multiple devices as a collective unified entity, or as groups of entities, within the whole structure.

The invention and evolution of the microprocessor has resulted in chips containing multiple cores, capable of running a large number of concurrent independent threads of execution, with separate "logical processors sharing the execution units of each core" [4, p. 10]. Moreover System on a Chip (SoC) devices are now available that combine CPU, memory, Graphical Processing Units (GPUs) and I/O on a single device [4]. Coupled with the development of operating systems, where each process can behave as its own virtual device—each containing its own stack, program counter and memory partition—these processes effectively function as logically separate entities [5]. As opposed to a physical multiplicity, where separate devices work together to form a unified function, we define these as "logical multiplicities" because separate threads of execution or processes—although coexisting on the same physical device—are logical entities that function together as a multiplicity. Extending this concept further, one could argue that two or more sketches running on the same device—similar to multiple patches concurrently running within the same instance of the program Max/MSP—could be considered as separate logical entities, and when communicating with one another, function as a logical multiplicity. Logical multiplicities are therefore abstract—they must be realised as either physical multiplicities, where the collective logic exists on more than one device; as purely logical multiplicities, in that one device performs all the work through separate threads of execution; or as a combination of the two.

One of the challenges of composing for multiplicities is deciding how to map data across multiple devices while treating the collection of devices as a single entity. The ability to simulate physical multiplicities through the use of logical multiplicities can significantly speed up the composition life-cycle. With the advent of the Internet of Things (IoT), however, there are a plethora of different device types of various computational power and connectivity capabilities. This in turn increases the level of complexity that increases the likelihood of errors due to protocol incompatibility, insufficient resources—such as computational power, energy supply or network bandwidth. Coupled with this, IoMusT devices are sometimes limited in their ability to access the internet due poor network coverage or firewall restrictions enforced by hosting institutions such as universities and public museums. The ability for composers to switch from purely logical multiplicities to physical ones without having to modify code to accommodate network based inter-device communication reduces frustration and promotes creative flux [6].

This paper introduces a mapping paradigm whereby shared variables can be mapped or bound using an object oriented strategy rather than a flat address-structured method; and how we implemented it in three separate creative works—an interactive planetarium software control interface [7], an interactive participatory tangible artwork [6], and an interface for capturing complex hand and finger movement during embroidery [8].

II. Context of Research

Many composers today are utilising the facilities afforded by network technologies that have enabled them to create agglomerate media artworks that involve multiple devices interacting with one another over networks. These works often

ISSN 2305-7254

requiring device intercommunication as well as sharing state information with one another, with composers often having to constrain their designs based on choices about how the work is distributed [9]. For example, choices are often made about whether computation is distributed among devices or managed on a central server, whether there will be network bandwidth reliability or bandwidth issues, and whether actual hardware will be available for a sufficient period of time beforehand to test and evaluate designs [2]. Although simulating a hardware device can be valuable for designing patches and testing they perform as an individual instrument [10], simulating multiple devices can be very challenging when considering that one would probably need to simulate device intercommunication as a part of this.

The research conducted in this paper has been as a part of the HappyBrackets platform [2]. HappyBrackets is a Java based creative coding environment where composers create sketches in Java, which are then compiled and sent to one or more Distributed Interactive Audio Devices (DIADs) for execution and performance. The DIAD was first introduced in 2014 as "an experimental design for creative sound and music performance and interaction using multiplicities of networked, portable computers" [11, p. 604]. DIADs contained a Raspberry Pi and an inertial measurement unit (IMU), and when handled by the audience and incorporated into the environment, they not only responded to user manipulation, but they also responded to one another. The focus of the project was the development of a reusable platform that allowed creators to easily develop interactive audio and easily deploy it to other devices [12].

Unlike many other embedded programming environments, such as Arduino, HappyBrackets facilitates running multiple sketches concurrently, allowing one to layer sketches on top of one another. Additionally, the HappyBrackets environment has a simulator feature, where it is possible to simulate a DIAD on the creative-coder's computer, send coded sketches to it, and run them as though they were physical DIADs. It is possible to simulate accelerometers and gyroscopes using sliders and text boxes. The ability to layer sketches on top of one another meant that it was theoretically possible to simulate more than one device, creating a logical multiplicity within a single virtual device. Communication between the devices in a physical multiplicity would require some sort of network communications, however, when the sketches run on same device as a purely logical multiplicity, then no network communication is required. Moreover, the concept of a multiplicity we were attempting to address is that the creative-coder should not need to be thinking about how to map messages across a network. Instead, they should be able to consider devices on the network as part of the multiplicity and being able to treat them as though they were all one big device, and unless the actual device specifics were pertinent to the application as a whole, this should be completely abstracted away from the coder. This, in turn, allows them to focus on the broader scope of the work. We approached the problem by providing a higher layer of abstraction where the creative coder could think in terms of logical relationships rather than physical ones.

III. Network Communication

Communication between devices is performed via some sort of network. Information is passed from one entity to another using a set of standards or rules that each party involved agrees to use during the communication process—this agreement is known as the *protocol*. For example, I cannot plug my Ethernet cable into a telephone socket and expect it to work because the protocols—physical, electrical, and software—are not compatible [13]. This requirement for compatibility of protocols is not limited to data communications, but any time two or more entities are required to cooperate for a common goal. An example of a physical protocol incompatibility would be attempting to plug the laptop computer power cable from Australia into the general power output socket in north America. Although the voltages are different, manufactures made it possible to use the 120V from North America in place of the 240V from Australia. However, although the different voltages are compatible, the physical connection protocols are different between the two, and so an adapter—a device that converts one protocol to another—is required to use it in the other country.

MIDI was introduced as a standardised method of sending data between musical devices in the 1980s [14]. The mapping paradigm was roughly based on the technology used by western electronic music composers. The channel addressing scheme was similar to sixteen channels of a mixing desk, with control messages generally sent to a particular channel. For example, each channel could be sent a message to play or stop a note, change the current instrument to play, or to send a control to modify the way a sound was playing. Additionally, some global message types were provided to facilitate synchronisation between devices [15]. Moreover, another feature provided in MIDI was the System Exclusive Message (SYSEX), which facilitated transmission of any other type of data that did not fit into the channel or global paradigm—a mechanism often used to develop complex patch editors for modifying configurable parameters in synthesis sound engines. Although MIDI became ubiquitous in the arts and entertainment industries, with "every PC from the early 1980s ... [having] either built-in or third-party MIDI interfaces available" [14, p. 67-68], the complexity of transmitting high-speed, high-resolution data over networks led to the development of Open Sound Control (OSC) in 1997 [16].

OSC introduced three significant paradigm shifts: an expandable and intuitive address space, the ability to provide more than one parameter or data type to a mapped point, and the ability to schedule a change of value some time in the future. Sadly, the last feature, which was "one of OSC's most important and interesting features had not been widely or correctly implemented....Three different and mutually incompatible uses of timetags have been employed over the years" [17, p. 116]. These three concepts can be abstracted away from the programmatic implementation employed by the OSC developers, in particular, the data structure used for OSC communication. These paradigm shifts effectively changed the way data was mapped and sent over networks. For example, a triple axis accelerometer no longer required three separate messages, but could be encapsulated in a single message. For example, sending the x, y and z values of an accelerometer could be done with OSC message `/accel` x, y, z. Similarly, if an additional device was added, one could just add to the OSC

address. For example, two accelerometers—`/accel/1` and `/accel/2`—could send their values shown in Fig. 1.

```
/accel/1 x, y, z
/accel/2 x, y, z
```

Fig. 1. Differentiation of physical accelerometers through the OSC address

Although these paradigm shifts have provided significant advantages to media art composers, artworks that require complex mapping by virtue of physical multiplicity [3] or by deep nested constructs means that these paradigms may no longer be adequate on their own. Difficulties can arise when mappings cannot be accommodated through a linear hierarchy or when addresses become complex. For example, a point from a Leap Motion frame capture might be */leap1/hands/left/fingers/ring/metacarpal*. Moreover, many researchers have noted that "different projects utilizing OSC are rarely protocol-compatible" [18], and consequently, have attempted to stretch the addressing structure to map the complex underlying data structures with implementations that include building OSC address trees [19], [20] and OSC address translation libraries [21], [22], [19].

Hunt, Wanderley and Paradis noted that mapping of control parameters through to synthesis is quite complex, and in order to perform other than simple one-to-one mappings, a separate abstraction layer is required between them. Moreover, they note "Now that we have the ability to design instruments with separable controllers and sound sources, we need to explicitly design the connection between the two. This is turning out to be a complex task" [23, p. 438]. This problem becomes more challenging when attempting to develop a strategy that facilitates both divergent and convergent mapping, facilitating one-to-many and many-to-one-relationships [24]. When examining the OSC example in Fig. 1, it is evident that although there is an intuitive name called `accel` for both, each point must be further defined through the addition of `1` or `2` in the OSC address in order to discriminate between the two bindings. This effectively means that the address has multiple semantic layers embedded into it. This addresses an additional layer of abstraction that was not evident when only looking at OSC messages in terms of simply address and data—data is actually passed from source to destination, which occurs at a lower layer in the OSI communications stack [25], [26]. Malloch, Sinclair and Wanderley have noted that "simple mappings in the semantic layer are in fact already complex and multi-dimensional" [24, p. 406], which resulted in the development of *Libmapper*—a software protocol that attempts to facilitate dynamic mapping by providing a separate semantic layer using OSC on a separate network port that provides a translation portal between devices on that network. Libmapper acts as a moderator between real-time OSC streams, negotiating mapping between input and outputs streams, such as sensors and synthesiser outputs [21]. The current developers of Libmapper are developing modifications to facilitate greater performance of communication between points on the same device by using memory pointers instead of the loopback port, with one developer stating: "Later we can optimize to avoid building the OSC messages at all but this would involve a lot of hacking since the handler deals with coordinating signal instances" [27].

The flat address model can also be restrictive when attempting to perform a single semantic function that might address multiple physical or logical parameters. For example, one of the features OSC provides to increase efficiency is the addition of address pattern matching in the OSC address [16]. For example a message to control a robot dancer `/dancer/lead/left/*` would match any address that starts with `/dancer/lead/left`. For example, the following two OSC addresses would both match that message:

```
/dancer/lead/left/hand
/dancer/lead/left/foot
```

The logical mapping is that of the left side of a single object—`dancer/lead`. A message of `/dancer/*/left/foot`, however, is different in that it controls the left foot of *all* dancers. A challenge occurs when we want to perform complex mapping coupled with an increased number of devices. If for example, the choreography required a particular three dancers to raise their left foot, the mapping would be somewhat more complex. The number of combinations required to target a single OSC address for every possible group of dancers is $(2^n - 1)$, where n is the number of dancers. Table I displays the number of address combinations required to target every combination of n number of dancers in `/dancer/`n`/left/foot`. It becomes obvious that as the number of devices and control points increase, the requirement for composers to map intuitive grouping name patterns becomes impractical. Although string comparisons are computationally expensive, with various developers attempting to improve the inefficiency of OSC address decoding through various techniques such as compressing the address length [28], adding new characters to indicate that an address can use a hashed lookup table [29] or embedding integers into addresses to increase lookup efficiency [30]; the real problem is the algorithmic inefficiency of encoding n in the flat address model. Algorithmic performance, also know as the "rate of growth, or order of growth" [31, p. 28], is the determining factor in how an algorithm's run time grows as the number of inputs increases, and is usually expressed using Big *O* notation [32]. The the algorithmic decoding complexity to create an intuitive name for each combination has a Big *O* value of $O(2^n)$, probably rendering it unsuitable for all but the smallest multiplicities. An alternative would be to use an integer as a parameter that defines which set of dancers to select—e.g. `/dancer/left/foot` $0 \Rightarrow (2^n - 1)$—reducing the complexity to $O(n)$, which would allow 32 dancers and require only 32 logical tests to account for 4 294 967 295 possible combinations.

TABLE I. NUMBER OF ADDRESS COMBINATIONS

Number Dancers	Number Combinations
2	3
3	7
4	15
8	63
10	1 023
20	1 048 575
32	4 294 967 295

Examination of an OSC message reveals that its semantic structure is almost identical to functions in the C programming language—they consists of a unique name and any number of

parameters. Although the C programming language facilitates encapsulation through the use of the *struct* keyword [33], it does not use object-oriented programming (OOP) constructs. "OOP isn't a language; it's the practice of architecting and the thought process behind it that leads to applications and languages being object-oriented" [34, p. 1]. This OOP method of thinking has resulted in various object based data exchange protocols being developed, such as XML [35] and JSON [36]. Instead of referencing every target address back to a root, sub-addresses were encapsulated as elements, referenced only to their immediate parent by containment. Moreover, omission of an element did not require padding for missing elements, as would be required by the array or list based encapsulation—similar to the message arguments implemented in OSC.

Similarly, we propose a new method of looking at shared parameters, treating them as objects rather than addressed functions. In the same way objects can have aliases, scope and behaviour within a program, we have attempted to treat data parameters across a networked multiplicity the same way. Rather than developing a new protocol, we developed a concept that strategically maps data parameters using aliases and scoping instead of a flat address-structure. Although we have implemented this strategy using text-based patching in Java using the HappyBrackets creative coding platform [2], the principles we propose could be also applied to graphical based programming languages. We will cover our implementation, focusing on the problems we found using Java based code, and how we addressed them.

IV. Dynamic Controls

We have implemented an event based API called *Dynamic Controls* that enables sharing of parameter values and events throughout the multiplicity using a single, simplified interface [37], whereby the same simple name can be used in multiple instances without clashing with other Dynamic Controls. This identical interface enables the composer to think in terms of the multiplicity as a single instrument regardless of where the code is running. Although initially created as a mechanism to pass a variable's state information to a graphical display, Dynamic Controls are extremely powerful in that they function as a message interconnection mechanism with other Dynamic Controls, whether on the same device or across a network [2], and can effectively function as shared parameters. Moreover, code is never translated to a streamed or packet based protocol unless it required to reach another device on the network (Libmapper developers are currently addressing how they may still be able to use the OSC addressing scheme, while at the same time, avoid the overhead of encoding and decoding OSC packets [27]).

Interaction between Dynamic Controls and their behaviour is defined through their *Value* type, *Name*, *Parent*, *Control Scope* and *Time*, shown in Fig. 2. The Control Scope attribute, which can be *unique, sketch, class, device, global,* or *target* scope, influences the communicative influence Dynamic Controls have on one another. In attempting to explain the various concepts, we will first provide analogies using simple Java code examples that illustrate the problem followed by conceptual solutions.

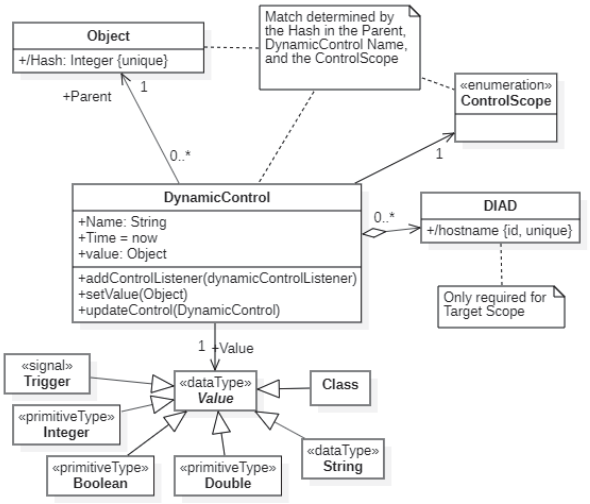

Fig. 2. Dynamic Control Model

A. Names and Aliases

One way of defining an entity is by giving it a symbolic lexical name. Effective variable or function names are often intuitive and consistent, making it easier for a person to understand its function and perform an operation on it. This, however, often results in identical names used in many different parts of a program, which would cause ambiguity if the entity being referenced was not specifically qualified. The ability to use the same name in various places is accomplished through the use of name spaces and scoping. Consider the following code:

```
class Gain{
  float val = 0;
  void setValue (float vol){val = vol;}
}
Gain g1 = new Gain();
Gain g2 = new Gain();

g1.val = 1;
g2.setValue (0.5);
```

There are two variables named *val*, one in each instance of the Gain objects—`g1` and `g2`. Although the value of `val` can be accessed directly from within the instance of the class—as shown in the function *setValue*—it cannot be accessed from outside the specific Gain object without first qualifying the instance of Gain to which is being referred. Changing the value of `g1.val` will have no effect on the value of `g2.val` and vice versa. This paradigm qualifies a data point at a single address in hierarchical terms, and the particular `val` referred to must be qualified by the Gain object that contains it.

It is, however, possible to create an alias of an object, and in this way, many different lexical names can be used to access the same object.

```
class Amplifier(){
  Gain gain;
  Amplifier(Gain g){ gain = g; }
  void setGain(float val){gain.setValue(val);}
}

Gain g1 = new Gain();
Gain g2 = g1;
```

```
Amplifier a1 = new Amplifier(g1);
Amplifier a2 = new Amplifier(g2);

a1.setGain(2); // everything is changed
```

In this example, the variables named `g1` and `g2` refer to the exact same object. Moreover, Amplifiers `a1` and `a2` share the same Gain object, and so changing `a1.gain.val` will also change `a2.gain.val`, `g1.val` and `g2.val` simultaneously. All of the variables are bound together through their association in that they share the same physical memory space because all the Gain objects are the same object throughout the code fragment. Also, although `g1` was created first, its state or ability to be assigned or propagated holds neither more nor less weight than the other three variables. The mapping between these variables is both convergent and divergent. In a sense, the *Name* attribute of the Dynamic Control functions as the variable name in the two code examples. Identical named controls can exist as disparate objects; or alternatively, they can function as aliases of one another, mirroring one another's actions. This relationship is based on the common association they have with a *parent* object, their *control scope* attribute, and the *data type* they present.

B. Association and Control Scope

In the same way the Amplifier objects from the earlier code example could modify each other's gain by virtue of their association to a common Gain object, Dynamic Controls are not only bound to one other through their common association with objects, but also by the type of relationship they share. The *Parent* attribute of the Dynamic Control functions as one of the association factors that determines whether controls are aliased or not. Dynamic Controls also have an attribute called *Control Scope*, which influences the communicative influence it has on other Dynamic Controls. We defined these scopes as *unique, sketch, class, device, target,* and *global*.

The unique scope treats each control as a disparate entity whose value is not shared with other Dynamic Controls. Retrieving values from a remote device, known as telemetry [10], is not a novel concept for musical instruments and has been employed in synthesizer patch editors for decades using MIDI system exclusive messages [38]. Although HappyBrackets is a text based programming environment, the ability for composers to set and read values using familiar interfaces like sliders, check boxes, buttons and text boxes can provide significant service to composers during composition or debugging a work [2], [39]. Unique scope controls allow users to create GUI interface that do not require communication with other Dynamic Controls.

Sketch scope enables controls to communicate with other controls with the same name and type that are linked to a specific instance of an object. If, for example, two floating point Dynamic Controls were both named "Pitch" and shared the same oscillator object as a parent, changing the value of one control would change the value of the other. Sketch control is particularly effective when layering sketches on top of one another, which can be extremely useful for simulating multiple devices in a multiplicity on a single computer.

Class scope associates controls based on the class type of the parent, creating a singleton association. In the same way that OOP languages like C++ and Java have static class members, where a single variable is shared by all objects of a class, class scope controls are associated based on the class type of the parent and not the instance. An example of where this type of control could be as a global setting on a device for a particular type of instrument. For example, if a class scope control named "grain size" was used to set the grain size of a specific type of granular synthesizer, varying the grain size would adjust the grain size on all instances of it. If another granular synthesizer type was created, one could create other class scope controls called "grain size" linked to this instrument, and the controls would not conflict with one another even though they share the same control name.

Device scope associates a control based on the device the sketch is running on, sending it to all matching controls on all sketches. An example for use of this type of control might be as a global pitch reference for all instruments on a particular device.

In addition to sharing variable parameters on a single device, Target and Global control scopes facilitates sharing values across the network to other devices. Global scope controls share their value with all devices, such as a global performance state. There can be unintended side effects when a large quantity of data is being shared over a network. An example of this was in a work where multiple roller skaters carried DIADs in their backpack in a live performance [40]. On the night of the performance, it was very difficult to hear the DIADs at the actual venue, so it became necessary to transmit the accelerometer and gyroscope values from the DIADs to another computer running similar patches so the sound could be synthesised based on the sensor values and played through a loudspeaker system. When global scope messages were used, the DIADs in backpacks were overloaded with incoming network messages, severely impacting on the ability of the DIADs to execute the required synthesis algorithm, rendering the sound unacceptable. On the night, data had to be transmitted using OSC messages directed only at the intended target. This limitation was resolved though the use of Target scope messages, where network messages can be directed to a specific set of devices instead of broadcasting the message.

Target scope controls are similar to global scope with the exception that specific devices can be targeted. For example, consider five DIADs all with target scope controls identically named "IMU", shown in Fig. 3. The controls in top three DIADs all target the bottom two DIADs, and consequently, setting a value in one of these top three will cause values to propagate to the bottom two. However, a message would not be sent to any of the top ones—not even on the same device. On the bottom left DIAD, setting a value in the left control would cause the value to be sent into itself, and to the DIAD to the right. Setting the value into the right hand control of that DIAD, however, would not propagate to any other controls because it does not have any targets set. Similarly, the control in the bottom right DIAD will not send its values to any other controls. One significant advantage of target scope controls is their ability to address devices specifically, thus preventing interruption to nodes for which messages are not intended. Target scope messages were used alongside global scope messages to implement a subscribe and push mechanism used in hand capturing interface.

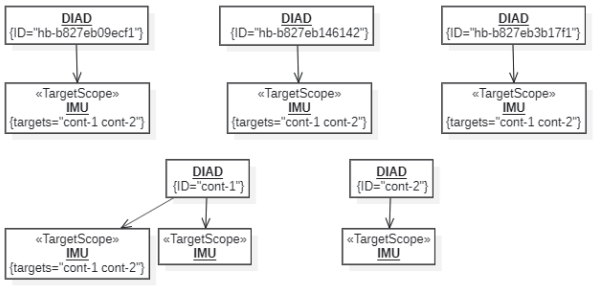

Fig. 3. Target Scope Controls

In addition to the control scope and the common parent association, the final mapping factor is the *type* of value being shared.

C. Value Type

One of the main paradigm shifts provided by OSC was the ability to map multiple data values and types to a single address. This concept is very similar to that of C programming language variadic functions, where they have a unique name—analogous to the OSC message name or address—and an indefinite number of arguments—analogous to the OSC message arguments [41]. One of the powerful advantages of this is that the user is not required to define the number of or types of arguments in advance. Whereas in General MIDI, the parameters for a note on message must be 4 bits, defining the MIDI channel, and 7 bits each for the note number and velocity. In OSC this restriction is removed completely—you can send whatever type or number of parameters that you want. This, however, is potentially problematic in that there is very high coupling between the sender and receiver due to a reliance on encoding and decoding the parameters concurrently on sender and receiver, and low cohesion due to universal data types.

Although the OSC namespace has a level of intuitiveness, this is lacking with OSC arguments. In the example `/accel/1` x, y, z, the order of arguments—i.e. x, y, z—is not necessarily intuitive. Logically, the absolute order of the arguments is irrelevant so long as both the encoding and decoding algorithms agree on the order and format of the data. This can result in cascading changes when the parameters are added, removed or re-ordered, resulting in the requirement to modify an unintuitive array index [42]. For example, consider a section of code where three parameters of an accelerometer are passed as a message.

```
// transmitter ******************
float x = 0, y = 0, z = 0;
OSCMessage message = new OSCMessage("/accel",
   new Object[]{x, y, z});

// receiver    ******************
float rx = (float) message.getArg(0);
```

The developer, after experimenting and refining their requirements, decides to add the accelerometer model number as the first parameter.

```
// transmitter ******************
String device = "LSM6DS33";
OSCMessage message = new OSCMessage("/accel",
  new Object[]{device, x, y, z});

// receiver    ******************
float x = (float) message.getArg(0); //
```

The receiver code is no longer valid, which means that the developers would need to find and modify all occurrences in the receiver code. The second problem is that the message arguments are not type safe. In this example, by virtue that the first argument is a String and not a float, the program would yield "runtime errors, which are difficult to detect and handle on distributed environments" [43, p. 45]. Our original API was

```
DynamicControlListener listener = new DynamicControlListener() {
    @Override
    public void update(DynamicControl dynamicControl) {
        String text = (String)dynamicControl.getValue();
        System.out.println("This is Control 1" + text);
    }
};
DynamicControl textControl1 = new DynamicControl("MyControl", "");
textControl1.addControlListener(listener);

DynamicControl textControl2 = new DynamicControl("MyControl", "");

textControl1.setValue("Hello World"); // send to listener and alias
```

Fig. 4. Original Dynamic Control API

based on the traditional event and listener paradigm, where the coder creates the control and adds a listener to it as shown in Fig. 4. The program is also able to access the variable directly from the control using the `getValue` function. Rather than providing an undefined number of variables as parameters, we only afford the user a single data parameter value, allowing a simple *get* and *set* facility. This parameter could be one of five primitive data types—*Trigger, Integer, Boolean, Double* and *Text*—or a complex *Class* data type for values or structures that cannot be defined using one of the primitives. In the example in Fig. 4, setting the value of `textControl1` causes its listener to execute its handler code, as well as setting the value inside variable `textControl2`. These two controls therefore effectively mirror one another. The problem with this API was

```
textControl1.setValue(false); // not type safe
String control2value = (String)textControl2.getValue(); // runtime exception
```

Fig. 5. Code showing get and set values are not type safe

twofold. First, although it was possible to access values directly from the controls themselves, notification of change required a listener—a separate entity to the control, resulting in high coupling between the control and the listener. This is evident by virtue of a required typecast to a string in the handler code. Second, there was no type safety facilitated with the value store in the control. Examination of the code in Fig. 5 reveals that it was possible to send a boolean message to the control, which will cause a run time error when cast to a string type. In a sense, this was equivalent to the existing paradigms—the user sends a universal data type value to an address and it is converted and actioned by a delegate. This can be resolved utilising function overloading within a composite pattern using type safe algorithmic skeletons [44], [43], [37].

One of the most significant changes from the C programming language to C++ was the addition of overloaded function names, whereby the same function name could be used with

different signatures, to perform different operations [45]. This is different to variadic functions or those that use a universal or undefined data types in their signature. Prior to function overloading, where the exact variable type is not known, the value passed down a chain of tests until it was handled [37]. In the following example, the value can be set using the function name `setValue` using either a float or an integer—the handler detects the data type and actions it.

```
class Gain {
  float value = 0;
  void setValue (Object param) {
    if (param instanceof Float) {
      value = ((float) param);
    } else if (param instanceof Integer) {
      value = (Integer) param;
    }
  }
}

Gain g = new Gain();
g.setValue(0.2f);
g.setValue(1);
g.setValue("0.4"); // this will do nothing
g.setValue(0.3); // this will do nothing
g.setValue(true); // this will do nothing
```

The problem with this method is that the API demands that the users implement every case that is required work. In the example given, although the first two calls will work, the last three will not because the implementation of the function `setValue` does not handle string, double or boolean types. Moreover, there is no error or warning given that the parameters are not valid for this function. A technique to overcome this is through syntactic overloading, which allows the same name to be used for variants of the same function [46]. Overloading the class function allows the user to call *setValue* with a string or numeric value.

```
public class Gain {
  float value = 0;
  void setValue(double val){value = (float)val;}
  void setValue (String val){doConvert(val);}
}
```

Moreover, boolean values would be rejected by the compiler as there is no overloaded function that would accept them. We address this through the use of algorithmic skeletons [44], shown in Fig. 6, whereby specialised classes handle only a single data type. In the code example shown in Fig. 7, `textControl_1` can only accept a String value. No typecast is required using the `getValue` function. Moreover, type safety is enforced in that it is not possible to set the control with a non-string type, shown when attempting to set its value to a boolean type.

Finally, we are able to use the same control name, `MyControl`, for multiple data types by using a different control type, as shown in Fig. 8. Even though the TextControl and BooleanControl share the same name and have matching control scope, they remain completely independent of one another. Moreover, by using a different parameter type we have practically overloaded the control name. In the same way variables are able to use the same variable name by using different scopes or associations, we are able to use different control scopes and associations to differentiate between controls with identical names throughout the multiplicity.

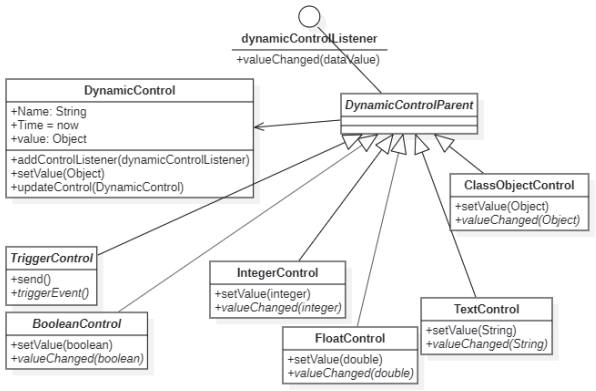

Fig. 6. Type safe algorithmic control skeleton

```
// Type textControl to generate this code
TextControl textControl_1 = new TextControl(this, "MyControl", "");

// Type textControl to generate this code
TextControl textControl_2 = new TextControl(this, "MyControl", "");

textControl_1.setValue("Hello World");
String value = textControl_2.getValue(); // this is the value in textControl_1

textControl_1.setValue(false); // compiler will not allow this
```

setValue (java.lang.String) in TextControl cannot be applied to (boolean)

Fig. 7. Enforcing type safety with parameters

D. Time

When two or more sections of code are required to produce time based deterministic outputs, such as playing sound at the same moment in time, a mechanism of synchronisation is required between two sections of code. Latency is the amount of time between a cause event, such as excitation or input stimulation, and the output from that event; whereas Jitter is the variability of latency [47]. Latency can be a fixed amount of time. For example, moving a sound source 3–4m away from a listener will introduce approximately 10ms of latency [48]. Latency, however, is also accumulative, whereby each input to a serial signal chain, such as DSP, adds additional latency. Research indicates that artists are generally better able to deal with higher fixed predictable latency far better than unpredictable lower ones [49], and that "it is generally possible to trade jitter for added latency" [48, p. 4]. Artists have generally conceded that there will be some sort of latency and jitter in real-time network based performances due to factors including network congestion, traffic, bandwidth, routing, propagation medium and processing time [50], [48], [26]. Similarly, although variables may share the same memory address on the one device, all sections of code that reference it—either directly or through an alias—are not necessarily able to action that change at the exact same moment in time. Although they are not impacted by the network latencies, they can be impacted by factors including threads waiting on locks, executing time based code and message queuing [7]. When setting a global or target scope control value, the latency between the control having its value set and it being read will appear greater when propagated across a physical network than those received on the same device, primarily due

```
// type booleanControl to generate this code
BooleanControl booleanControl = new BooleanControl(this, "MyControl", false);
booleanControl.setValue(true);
```

Fig. 8. Overloading the "MyControl" name with BooleanControl

to the addition of network latencies and jitter. Moreover, the unpredictable nature of network jitter exacerbates the problem of cross-device variable sharing synchronisation [51], [52]. To effect an API that is agnostic to whether parameters are on a local device or over a network, an identical interface must be presented to the coder whether the control is local or over the network. One solution to create a more deterministic relationship between device-shared variables to hide the difference in latency by triggering or scheduling changing variable states and overlapping network jitter within the scheduled time [53].

1) Variable Event Triggering: The value of a Dynamic Control can be accessed directly through a *getter* function, eg `myval = control.getValue();`, or through event notification. "Although event-based software integration is one of the most prevalent approaches to loose integration, no consistent model for describing it exists." [54, p. 378]. One of the most common methods of describing it is through the event and listener metaphor. In this metaphor, an object receives a message, however, the responsibility for acting on that change is delegated to another object. Research indicates that the event listener metaphor is not intuitive, with one researcher stating that the "listening metaphor is false" [55, p. 76]. In our original API, shown in Fig. 4, separate listener was responsible for performing an action when the value of `textControl1` was changed. A limitation with this pattern is that the semantic relationship between these two are not contained and manipulated as a single unit [56]. A significant change from functional to OOP was the treatment of objects as separate entities that could only be accessed by sending them messages, resulting in the object executing the method associated with that message [57]. In the given example, however, control was passed to the listener, using the control as a parameter. A more cohesive solution would be to have the objects perform the function directly. We implemented this inside the type safe skeleton, where there is no separate listener. If the user requires the object to perform some action as a result of the value being changed, they can overload the `valueChanged` function as shown in Fig. 9. In the code example, one can see that although the top control returns no assigned variable when created, it is modified by the `textControl_1` having its values set, which causes "Hello World" to print via the `valueChanged` handler.

```
new TextControl(this, "MyControl", "") {
    @Override
    public void valueChanged(String control_val) {
        // Write your DynamicControl code below this line
        System.out.println(control_val);
        // Write your DynamicControl code above this line
    }
};// End DynamicControl textControl_2 code

// Type textControl to generate this code
TextControl textControl_1 = new TextControl(this, "MyControl", "");
textControl_1.setValue("Hello World");
```

Fig. 9. Integrated event notification

2) Scheduled parameter value changes: Synchronisation of two or more devices that each operate with an internal clock generally require some sort of synchronisation. Various synchronisation techniques and algorithms have been developed throughout the years that enable various devices to determine a common point of time reference. In our API, the method of synchronisation is not important—there is no requirement for an absolute reference time outside the domain of the network and the code running it. Our API uses a double precision point time representing the number of milliseconds from a set epoch. To set when the value is meant to change, the control value is set using the time the change is supposed to occur. Fig. 10 shows two *FloatControl* objects with global scope. The top section of code could be running on any number of DIADs, whereas the bottom section of code symbolised code that is meant to set these controls 10s into the future. Fig. 11 displays the sequence of events that take place. First, the user sets the value of the control, however, a scheduled time for the event is added as a parameter. The control on the local device registers with the scheduler on the local device to call back at the set time. Next, the message is passed across the network to any other matching controls, which notify the scheduler on that device to notify them when the defined time has arrived. When the time has arrived—10s after the user code made the call—the schedulers on each device fire simultaneously, causing all matching Dynamic Controls on that device to set their value, triggering the `valueChanged` event if overridden. A notable feature is that the only change in the API between scheduled and non-scheduled parameter changes is the addition of the optional time parameter in the `setValue` function.

```
// This is on both devices
new FloatControl(this, "TimeCriticalCode", 0) {
    @Override
    public void valueChanged(double control_val) {
        // Write your DynamicControl code below this line
        doSomeTimeCriticalAction(control_val);
        // Write your DynamicControl code above this line
    }
}.setControlScope(ControlScope.GLOBAL);// End DynamicControl code floatControl
// write your code above this line
```

```
// This is the code that will trigger event
FloatControl eventSender = new FloatControl(this, "TimeCriticalCode", 0)
        .setControlScope(ControlScope.GLOBAL);

double current_time = HB.getSchedulerTime();
double execution_time = current_time + 10000; // this is ten seconds from now
eventSender.setValue(2.0, execution_time);
```

Fig. 10. Global scheduled message

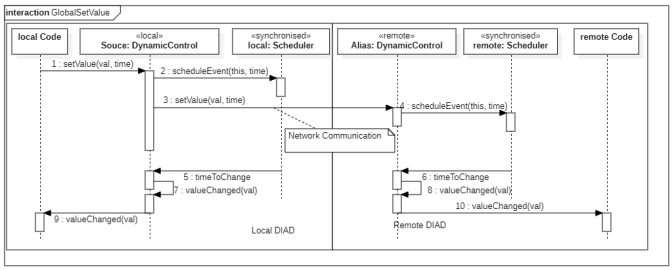

Fig. 11. Scheduled value sequence

E. Underlying Network

Although the underlying communication protocol between devices on the network is outside the scope of the API, we feel it is relevant to briefly describe what actually occurs. The underlying protocols are not important so long as the encoding and decoding algorithms compliment one another. Although we are using OSC as our underlying protocol, this is merely for convenience. Instead of using the OSC address as a mechanism for addressing data or parameters, we using it to address the functionality required of the message. There are four message types for communicating between DIADs using the API—update value message, global scope message, target scope message and device name message. All the parameters that define the control value are packed as OSC message parameters. Similarly, we are able to graphically display the controls by sending messages to GUI interface based on a unique numerical key that matches a control on a device, similar to the way a program accesses an underlying device through a handle. The messages are: 1) a control exists, 2) a control has been removed, or 3) a request from the GUI to send all control handles. In short, the OSC addresses only act as a short group of function names. Some of the OSC parameters, however, act as virtual function pointers within the Dynamic Control code. OSC could be quite easily replaced as the underlying communications protocol as the API is not dependent upon it. The new intercommunication facilities provided with MIDI-2.0 facilitate property exchange and two way communication between devices and include features such as compression and complex parameter encoding [58]. If we decide to change the underlying protocol, this would not affect the API we developed because this is delegated to the underlying layers and is invisible to the API user.

V. Example usages

We briefly describe how three separate creative works made use of various features of the Dynamic Control API—an interactive planetarium software control interface [7], an interactive participatory tangible artwork [6], and an interface for capturing complex hand and finger movement during embroidery [8].

A. Planetarium Software Interface

The first implementation of our API was used to control planetarium software from a DIAD. This was first realised as a interactive sonified virtual spacecraft simulator game, where the goal was for players to navigate to various planetary or stellar objects by manipulating a DIAD, which in turn controlled a planetarium software display on a computer. The DIAD, a battery powered Raspberry Pi Zero running only HappyBrackets, contained an IMU for input and a speaker for audio output. The computer, however, ran both HappyBrackets and *Stellarium* planetarium software. When the user manipulated the DIAD, IMU data was sent via a global control—e.g. z–axis shown in top part of Fig. 12 as "LR Movement"—to the computer running the receive sketch, shown at the bottom. During the development stages, it was easier to manually modify the values with the HappyBrackets GUI than it was to hold the DIAD and change the value, shown in Fig. 13, enabling us to see how the planetarium software would respond to simulated values. Switching between DIAD control and GUI control required no changes to the code whatsoever—the DIAD sketch could be stopped and restarted in less than a second, enabling us to maintain creative flux. Moreover, it was also possible to view the live values being sent by the DIAD in the GUI, again without changing any code. The effect was control addresses never required changing when when switching between development and running with the DIADs.

```
FloatControl lrSender = new FloatControl( parent_sketch: this,  name: "LR Movement",  initial_value: 0)
        .setControlScope(ControlScope.GLOBAL);

/** type accelerometerSensor to create this. Values typically range from -1 to + 1 **/
new AccelerometerListener(hb) {
    @Override
    public void sensorUpdated(float x_val, float y_val, float z_val) {
        upSender.setValue(x_val);
        lrSender.setValue(z_val);
    }
}.setRounding(2);                                          Code in DIAD for sending
```

```
new FloatControl( parent_sketch: this,  name: "LR Movement",  initial_value: 0) {
    @Override
    public void valueChanged(double control_val) {
        LRMovementAmount = control_val;
        synchronized (lrMoveSynchroniser){                 Code on computer for reading
            lrMoveSynchroniser.notify();
        }
    }
}.setDisplayRange( minimum: -1,  maximum: 1, DynamicControl.DISPLAY_TYPE.DISPLAY_ENABLED_BUDDY)
        .setControlScope(ControlScope.GLOBAL);
```

Fig. 12. Global control message between DIAD and Computer

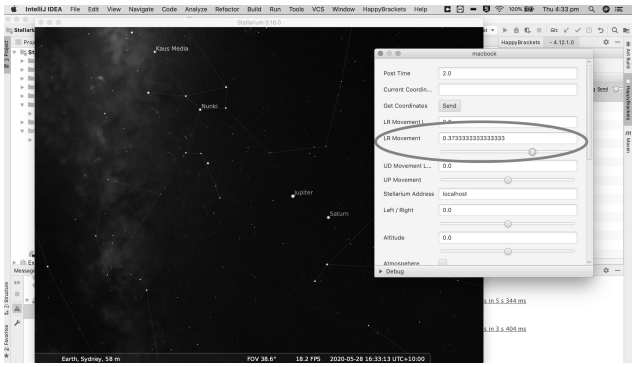

Fig. 13. Manipulating control values in GUI

B. Participatory Artwork

The next example of Dynamic Control usage was a work that used thirteen DIADs, where ten of the DIADs ran identical sketch code [6]. Three of the DIADs were interactive sonic balls, while the remaining ten were stationary devices. First, a global scope control was used to share the global state of the performance [6]. In the work, there was no master controller that determined what the current performance state should be. In one performance state, for example, two balls were independently rolled by members of a participatory audience. When one of the balls determined that the ball had been rolled sufficiently, it changed the performance state for the whole multiplicity by sending the global scope message.

It is often convenient to simulate embedded devices on the computer used for composition rather than uploading code to the embedded device. The device simulator in HappyBrackets is a virtual DIAD that runs identical code to a physical DIAD, however, it facilitates simulation of sensor devices, such as accelerometers and gyroscopes, by using sliders [2]. Using the simulator would eliminate the necessity to upload code to the DIAD and physically roll it when performing various tests during the composition refinement stage. Moreover, it reduced

the number of times the composer would need to recharge batteries on each of the devices during the composition stage. Although HappyBrackets does not enable one to run more than one virtual device of the computer, it is possible to simulate multiple devices by layering sketches on top of one another, creating a pure logical multiplicity. Instead of sending a single sketch to thirteen different DIADs during the composition phase of the work, the required sketches were sent multiple times to simulator—three sketches that utilised IMUs and ten for static sonification—thus simulating the entire multiplicity [6]. Global scoped controls facilitated communication between all sketch instances, with no changes required to any of the code when changing to sketches running on the networked devices. Moreover, it was possible to add or remove more DIADs to the multiplicity without requiring any address mapping changes.

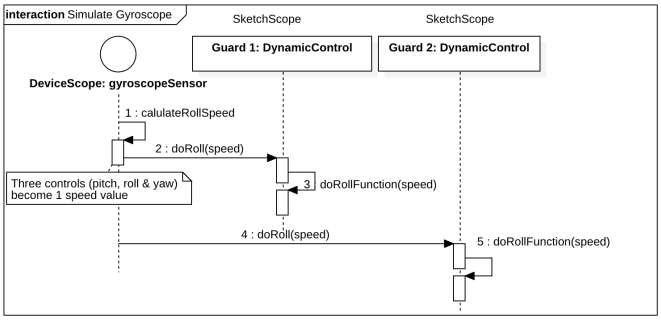

Fig. 14. Device and Sketch ControlScope

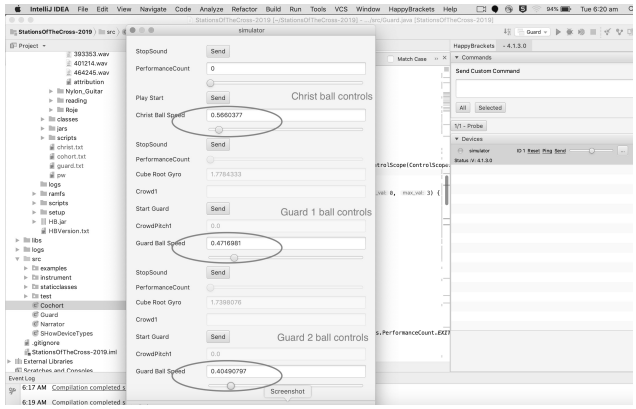

Fig. 15. Device and Sketch ControlScope

Simulating the three balls would require three sets of gyroscopes, however, the virtual device simulator only has one set of gyroscope controls. It is possible, however, to simulate these as separate controls for each device by using the gyroscope average values through a *SketchScope* control. Sensor simulator controls would have a *DeviceScope*, in that the values are transmitted to all sketches on the device. The *SketchScope* control only transmits its value to other DynamicControls in that same sketch. This effectively enables one to individually simulate an independent IMU for each sketch on the virtual device by connecting the *DeviceScope* control to the *SketchScope* control, shown in Figure. 14. Fig. 15 shows how the three interactive DIADs were simulated in the HappyBrackets GUI.

C. Complex Hand Movement Interface

The final work we present using our Dynamic Control API was for collecting complex hand and finger movements during embroidery. The primary focus of the research was to capture the complex hand gestures of traditional crafts, which often take a lifetime to acquire, to gain new perspectives on complex hand skills and ways of sharing the traditional wisdom embodied in these practices through development of a digital audio-visual interface [8]. The capture technique involved collecting synchronised multi-modal data consisting of three PixiCam cameras, a LeapMotion detector, and raw video capture. Each PixiCam captured continuous frames of the embroiderer's individual finger positions effected through the use of coloured nail polish on their hands, mapping various colour combinations to fingers. The three PixiCams, facilitating three axes, were all connected to the one Raspberry Pi through an I2C bus and grouped as a timestamped three dimensional frame. The LeapMotion detector uses a stereo infra-red camera to capture a three dimensional image of the person's hands, and using the LeapMotion software, able to determine pitch, roll and yaw, as well as finger extensions and positions. Similarly, the LeapMotion data was collated and timestamped as frames of data. At no time with the PixiCam or LeapMotion would it be possible to predetermine how much data would be in any frame, shown in Fig. 16.

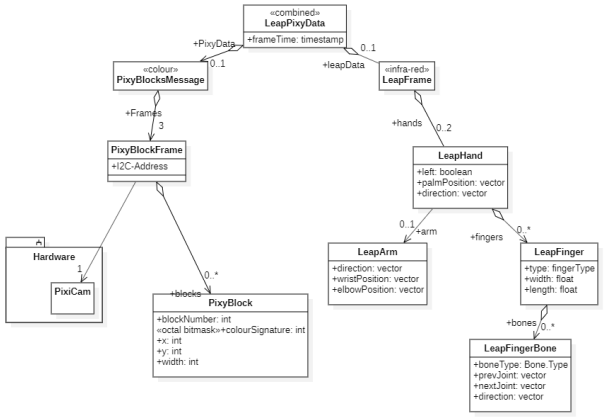

Fig. 16. Embroidery Data Frames

The Raspberry Pi had to poll all three PixiCam devices and send the data to the capturing computer approximately every 20ms. Instead of using a global control to send the data, coupled global and target scope pair were used in a publish-subscribe pattern [37, p. 293]. The computer required to collate all the data subscribes by sending a global text message called "SendPixy" with it's device name, shown in the top section of Fig. 17. In the bottom section of the code, the device connected to PixiCam receives the subscription message, adds the device name as a target to the `pixyBlockSender` *ClassObject-Control* with control name of "pixycam.PixyBlock". The code simply has to call `setValue` with the PixyCam data, located in the variable `pixyMessage`. In the top section of the code, the control named "pixycam.PixyBlock" receives the code and

actions it. Using the Target Scope control prevents the Pi from receiving messages from itself, only directing them to where they are required.

```
// in Data collator
TextControl pixyRequester = new TextControl(this, "SendPixy", "").setControlScope(ControlScope.GLOBAL);
pixyRequester.setValue(Device.getDeviceName());

// Type classObjectControl to generate this code
new ClassObjectControl(this, "pixycam.PixyBlock", PixyBlocksMessage.class) {
    @Override
    public void valueChanged(Object object_val) {...}

}.setControlScope(ControlScope.TARGET);// End DynamicControl pixcamBlockReceiver code
```

```
// In Pi connected to PixiCam
PixyBlocksMessage pixyMessage = new PixyBlocksMessage(NUM_PIXY_CAMS);

ClassObjectControl pixyBlockSender = new ClassObjectControl(this, "pixycam.PixyBlock",
        PixyBlocksMessage.class).setControlScope(ControlScope.TARGET);

new TextControl(this, "SendPixy", "") {
    @Override
    public void valueChanged(String control_val) {
        pixyBlockSender.addControlTarget(control_val);
    }
}.setControlScope(ControlScope.GLOBAL);// End DynamicControl textControl code

// this will not send until a target has been added
// This is in a loop where pixyMessage is filled
    pixyBlockSender.setValue(pixyMessage);
```

Fig. 17. PixiCam messages sent via TargetScope control

The data is joined with LeapMotion frames as JSON data and stored to disk. Our intention is to synchronise the data with the raw video and then enter into a machine learning algorithm to learn various stitching patters. With the API, there is no requirement to encode or decode the complex data structure as it is completed in the lower layers, outside the scope of what the creative coder is required to do.

VI. Conclusion

We presented an alternate paradigm of sharing parameters throughout a multiplicity whereby variables are treated objectively rather than functionally. The existing models of defining parameters were based on a flat addressing scheme, and although were suitable to procedural or functional programming paradigms, we treated parameters as objects that can have scope, aliases, and could be manipulated based on their logical placement rather than their physical location. Instead of moving outside the OSC specification, we have altered the way we view and implement OSC messages in our software, while still maintaining OSC 1.0 compatibility when communicating between devices over the network. The ability to provide different scopes for parameters or functions could be potentially employed to other programs. For example, the send and receive pair in Max allows passing of messages between patches without requiring patch cables [59]. They currently facilitate sending messages globally based on a single name, although it can be restricted to a single device through a generated ID using Live [60]. Opening two patches in Max that have the same send and receive names could cause unintentional cross communication between the two. The facility to assign a level of scope and association would prevent this type of cross-talk, while allowing greater flexibility in mapping.

The ability to send and/or access complex data types as complete entities rather than lists of parameters promotes data abstraction and encapsulation, allowing greater flexibility through modular architecture as underlying data structures can change during the lifestyle or evolution of a computer based composition. Additionally, the facility to define data accessibility, and the ability to reuse human readable names based on a variable's scope is a common feature of most programming languages. This paradigm has been extended in that scoping a variable can be dynamically bound or addressed to specific objects, class types, devices or globally on an entire network.

Finally, we presented three works that utilised this new mechanism of parameter sharing throughout the multiplicity, showing how implementing control scope and unified data enabled composers to think about their parameters as logical entities, agnostic to the network, and treated as though they existed on the same device.

Acknowledgment

We acknowledge the work of Patricia Flanagan and Saurabh Srivastava for their work in embroidery capture project. We also acknowledge the support of the Australian Research Council through their Linkage Grant (LP180100151).

References

[1] L. Turchet, C. Fischione, G. Essl, D. Keller, and M. Barthet, "Internet of musical things: Vision and challenges," *IEEE Access*, vol. 6, pp. 61994–62017, 2018.

[2] A. Fraietta, O. Bown, S. Ferguson, S. Gillespie, and L. Bray, "Rapid composition for networked devices: HappyBrackets," *Computer Music Journal*, vol. 43, no. 2-3, pp. 89–108, 2020.

[3] O. Bown and S. Ferguson, "Understanding media multiplicities," *Entertainment Computing*, vol. 25, pp. 62–70, 2018.

[4] W. Stallings, *Operating systems: internals and design principles*, Prentice Hall, Upper Saddle River, NJ, 2009.

[5] U. Vahalia, *UNIX internals: the new frontiers*, Prentice Hall, Upper Saddle River, N.J., 1996.

[6] A. Fraietta, "Transient relics: Temporal tangents to an ancient virtual pilgrimage," in *Fourteenth International Conference on Tangible, Embedded, and Embodied Interaction*, New York, NY, USA, 2020, TEI '20, ACM.

[7] A. Fraietta and O. Bown, "Creating a sonified spacecraft game using HappyBrackets and Stellarium," in *Proceedings of the 17th Linux Audio Conference (LAC-19)*. CCRMA, Stanford University, USA, 2019, pp. 1–7.

[8] P. J. Flanagan and A. Fraietta, "Tracing the intangible: The curious gestures of crafts' cultural heritage," in *Adjunct Proceedings of the 2019 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2019 ACM International Symposium on Wearable Computers*, New York, NY, USA, 2019, UbiComp/ISWC '19 Adjunct, p. 49–52, Association for Computing Machinery.

[9] O. Bown and S. Ferguson, "Creative media+ the internet of things= media multiplicities," *Leonardo*, vol. 51, no. 1, pp. 53–54, 2018.

[10] A. Fraietta, "The smart controller workbench," in *Proceedings of the 2005 conference on New interfaces for musical expression*, Vancouver, BC, Canada, 2005, University of British Columbia, pp. 46–49.

[11] O. Bown, L. Loke, S. Ferguson, and D. Reinhardt, "Distributed interactive audio devices: Creative strategies and audience responses to novel musical interaction scenarios," in *International Symposium on Electronic Art*. ISEA, 2015, pp. 604–607.

[12] O. Bown and S. Ferguson, "A musical game of bowls using the diads," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2016, pp. 371–372.

[13] J. Axelson, *Serial port complete: COM Ports, USB virtual COM ports, and ports for embedded systems*, Lakeview Research, 2007.

[14] P. Doornbusch, "Early hardware and early ideas in computer music: Their development and their current forms," in *The Oxford Handbook of Computer Music*. Oxford University Press, New York, NY, USA, 2009.

[15] J. Pressing, *Synthesizer performance and real-time techniques*, AR Editions, Inc., 1992.

[16] M. Wright and A. Freed, "Opensound control: a new protocol for communicating with sound synthesizers," in *Proceedings: International Computer Music Conference 1997, Thessaloniki, Hellas, 25-30 september 1997*. The International Computer Music Association, 1997, pp. 101–104.

[17] A. Freed and A. Schmeder, "Features and future of open sound control version 1.1 for nime," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Pittsburgh, PA, United States, 2009, pp. 116–120.

[18] "Libmapper motivation," Web: http://libmapper.github.io/about.html.

[19] T. Place, T. Lossius, A. R. Jensenius, and N. Peters, "Addressing classes by differentiating values and properties in OSC," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Genoa, Italy, 2008, pp. 181–184.

[20] M. Wright, A. Freed, and A. Momeni, "Opensound control: State of the art 2003," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Montreal, Canada, 2003, pp. 153–159.

[21] J. Malloch, S. Sinclair, and M. M. Wanderley, "Libmapper:(a library for connecting things)," in *CHI'13 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2013, pp. 3087–3090.

[22] I. Bergstrom and J. Llobera, "OSC-namespace and OSC-state: Schemata for describing the namespace and state of OSC-enabled systems," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, London, United Kingdom, June 2014, pp. 311–314, Goldsmiths, University of London.

[23] A. Hunt, M. M. Wanderley, and M. Paradis, "The importance of parameter mapping in electronic instrument design," *Journal of New Music Research*, vol. 32, no. 4, pp. 429–440, 2003.

[24] J. Malloch, S. Sinclair, and M. M. Wanderley, "A Network-Based Framework for Collaborative Development and Performance of Digital Musical Instruments," in *Computer Music Modeling and Retrieval. Sense of Sounds*, R. Kronland-Martinet, S. Ystad, and K. Jensen, Eds., Berlin, Heidelberg, 2008, pp. 401–425, Springer Berlin Heidelberg.

[25] F. Nazir and A. Seneviratne, "Towards mobility enabled protocol stack for future wireless networks," *Ubiquitous Computing and Communication Journal*, vol. 2, no. 4, pp. 65–79, 2007.

[26] A. Fraietta, "Open sound control: Constraints and limitations," in *Proceedings of International Conference on New Musical Interfaces for Music Expression (NIME-2008)*. Genova, Italy, Juns 2008, pp. 19–23.

[27] "Dot mapper: Use pointers for internal mappings," Web: https://groups.google.com/forum/?utm_source=footer#!msg/dot_mapper/0Qst4QCCJ5g/mzj3g6neAwAJ.

[28] J. Kleimola and P. J. McGlynn, "Improving the efficiency of open sound control with compressed address strings," in *Proceedings of the 8th Sound and Music Computing Conference (SMC)*, 2011.

[29] R. B. Dannenberg and Z. Chi, "O2: Rethinking open sound control," in *Proceedings of the International Computer Music Conference*, 2016, p. 494.

[30] J. Kleimola et al., *Nonlinear abstract sound synthesis algorithms*, Aalto University, 2013.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, MIT press, 2009.

[32] I. Chivers and J. Sleightholme, *An Introduction to Algorithms and the Big O Notation*, pp. 359–364, Springer International Publishing, Cham, 2015.

[33] B. W. Kernighan and D. M. Ritchie, *The C programming language*, 2006.

[34] B. Smith, *Object-Oriented Programming*, pp. 1–25, Apress, Berkeley, CA, 2011.

[35] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (XML)," *World Wide Web Journal*, vol. 2, no. 4, pp. 27–66, 1997.

[36] T. Bray et al., "The javascript object notation (json) data interchange format," 2014.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education, Reading, MA, 1994.

[38] D. M. Huber, *The MIDI manual: a practical guide to MIDI in the project studio*, Focal Press, Waltham, MA, USA, 2012.

[39] O. Bown, A. Fraietta, S. Ferguson, L. Loke, and L. Bray, "Facilitating creative exploratory search with multiple networked audio devices using happybrackets," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Porto Alegre, Brazil, June 2019, pp. 286–291, UFRGS.

[40] L. Loke, A. Fraietta, P. Crawley, A. Winston, and M. Leete, "Sonic sk8er," Performance, Aug 2019.

[41] M. Blume, M. Rainey, and J. Reppy, "Calling variadic functions from a strongly-typed language," in *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. ACM, 2008, pp. 47–58.

[42] M. Fowler, "Patterns of enterprise application architecture," 2003.

[43] D. Caromel, L. Henrio, and M. Leyton, "Type safe algorithmic skeletons," in *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, 2008, pp. 45–53.

[44] M. I. Cole, *Algorithmic skeletons: structured management of parallel computation*, Pitman London, 1989.

[45] M. A. Ellis and B. Stroustrup, *The annotated C++ reference manual*, Addison-Wesley, 1990.

[46] B. Meyer, "Overloading vs. object technology," *Journal of Object Oriented Programming*, vol. 14, no. 4, pp. 3–7, 2001.

[47] M. Wright, R. J. Cassidy, and M. Zbyszynski, "Audio and gesture latency measurements on linux and osx.," in *ICMC*, 2004.

[48] N. Lago and F. Kon, "The quest for low latency.," in *ICMC*, 2004.

[49] R. H. Jack, T. Stockman, and A. McPherson, "Effect of latency on performer interaction and subjective quality assessment of a digital musical instrument," in *Proceedings of the audio mostly 2016*, pp. 116–123. 2016.

[50] R. Goonatilake and R. A. Bachnak, "Modeling latency in a network distribution," *Network and Communication Technologies*, vol. 1, no. 2, pp. 1, 2012.

[51] L. Cheng and C.-L. Wang, "Network performance isolation for latency-sensitive cloud applications," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1073–1084, 2013.

[52] L. Cheng, C.-L. Wang, and S. Di, "Defeating network jitter for virtual machines," in *2011 Fourth IEEE International Conference on Utility and Cloud Computing*. IEEE, 2011, pp. 65–72.

[53] X. Guo, J. Dai, L. Li, Z. Lv, and P. R. Chandra, "Latency hiding in multi-threading and multi-processing of network applications," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. IEEE, 2007, pp. 270–279.

[54] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise, "A framework for event-based software integration," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 4, pp. 378–421, 1996.

[55] W. W. Milner, "A broken metaphor in Java," *ACM SIGCSE Bulletin*, vol. 41, no. 4, pp. 76–77, 2010.

[56] J. Rumbaugh, "Relations as semantic constructs in an object-oriented language," in *Conference proceedings on Object-oriented programming systems, languages and applications*, 1987, pp. 466–481.

[57] P. America, "Inheritance and subtyping in a parallel object-oriented language," in *European Conference on Object-Oriented Programming*. Springer, 1987, pp. 234–242.

[58] AMEI and MMA, "Common rules for MIDI CI property exchange Version 1.0," Web: https://www.midi.org/midi2, 2020.

[59] M. Puckette, "Combining event and signal processing in the max graphical programming environment," *Computer music journal*, vol. 15, no. 3, pp. 68–77, 1991.

[60] C. '74, "send: Send messages without patch cords," Web: https://docs.cycling74.com/max8/refpages/send.