Information Architectures for Personalized Multimedia

by

Klee Dienes

S.B., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
May 1993

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

© Massachusetts Institute of Technology, 1995.
All Rights Reserved.

Author ....
Program in Media Arts and Sciences
May 26, 1995

Certified by ....
Walter Bender
Associate Director of Information Technology, Program in Media Arts and Sciences
Thesis Supervisor

Accepted by ....
Stephen A. Benton
Chairperson, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

MASSACHUSETTS INSTITUTE
JUL 06 1995
LIBRARIES

Information Architectures for Personalized Multimedia

by

Klee Dienes

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on May 26, 1995
in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE

Abstract

As multimedia delivery systems become more sophisticated, they increasingly demand the use of specialized data formats to represent content. The `dsys` and `dtype` libraries and protocols provide a simple mechanism to represent, transmit, manipulate, and evaluate structured data and embeddable Scheme programs in a network- and machine- independent manner. Both as a hybrid programming environment for Scheme and C++, and as a highly extensible environment for distributed computation, the `dsys` and `dtype` libraries and protocols provide a strong basis for interactive multimedia applications.

Thesis Supervisor: Walter Bender
Title: Associate Director of Information Technology, MIT Media Laboratory

This work was supported by the News in the Future research consortium.

Information Architectures for Personalized Multimedia

by

Klee Dienes

The following people served as readers for this thesis:

Thesis Reader . . . .

Andrew Lippman
Associate Director, MIT Media Laboratory

Thesis Reader . . . .

Kenneth B. Haase
Assistant Professor, Program in Media Arts and Sciences

Table of Contents

1 Introduction 6

2 Fundamentals 6

2.1 The Dtype Transport Protocol 7
2.1.1 Binary Format 7
2.1.1.1 The Type Field 7
2.1.1.2 Representation Formats 8
2.1.1.3 Fundamental Types 10
2.1.2 Interactive Protocols 12
2.1.2.1 Connection Protocol 12
2.1.2.2 Command Protocol 13
2.1.2.3 Notification Protocol 14
2.1.2.4 Example: Garden CD Server 15
2.1.3 The Dtype Object System 15
2.1.4 Conclusion 20
2.2 Dsys and Derived Iostreams 20
2.2.1 Dsys 21
2.2.2 Derived Iostreams 21
2.2.3 Dsys Pollblocks / Servers 22
2.2.4 Conclusion 22
2.3 The Dtype Distributed Computation Environment 22
2.3.1 Remote Object System 22
2.3.2 Meta-Object Protocol 24
2.4 The Dtype Class Library 24
2.4.1 Data Structures 24
2.4.2 The Dtype Method System 26
2.5 The Dtype Scheme Evaluator 27
2.5.1 Hybrid Programming 27

3 Applications to Multimedia 28

3.1 Object Layer 29
3.2 Control Layer 32
3.3 Display Layer 33
3.3.1 MBPlayer 33
3.3.2 HtmlViewer 34

3.4 Example Application: Media Bank Web Browser 34

4 Conclusion 38

4.1 Caveats 38
4.2 Acknowledgements 38
4.3 References 39

5 Appendix A 40

1 Introduction

Current designs for multimedia information systems demand a tight coupling between presentation applications and the data they display. Standardization efforts address this problem by trying to reduce the number of formats and representations available. Given the wide variety of data necessary for a complete multimedia information system, it is unlikely that any small set of data formats will suffice for all possible applications. Supporting all the specialized formats for application data makes client development an increasingly tedious process.

Simple transcoding protocols like that of HTTP and MIME are helpful only in certain limited situations: MIME requires the creator of a document to decide in advance all of the viewing formats a document will support, and HTTP requires the assumption of an intelligent process serving the data and provides only a limited format negotiation mechanism. Instead, new ways of representing multimedia data are required that allow the use of custom data representations by applications that do not know about them in advance.

I propose the use of a Scheme-based data representaion for network-based information agents and multimedia objects. In this thesis, I develop the `dsys` and `dtype` protocol and distributed computing environment, a set of protocols and C++ libraries that support a distributed computation environment based on the exchange and remote evaluation of Scheme-like data structures.

In the context of the `dsys` and `dtype` libraries, I go on to develop a variety of multimedia and news-based network services and multimedia applications. Although the work will be presented in the context of the dtype libraries for increased concreteness, it is equally applicable to any highly-extensible scheme library supporting distributed computation and embeddable evaluation.

2 Fundamentals

The dtype library has four main layers, each of which can be used independently of the layers above it. First and most important is the the dtype transport protocol. This protocol is used to provide a compact and efficiently parseable representation of Scheme data structures for network transmission and binary storage. Associated with this layer are the dsys and derived iostream libraries, which provide support for stream processing functions from C++. The second layer is the dtype class library, a collection of Scheme data structures for C++. The third is the dtype evaluator library, a Scheme evaluator and extension language based on the dtype

data structures. The final layer is the dtype distributed computation environment, a remote object system based on the dtype protocol.

2.1 The Dtype Transport Protocol

The dtype transport protocol provides a way to represent Scheme data structures in a binary format. The transport representation of a Scheme data type need have no relation to the representation for that type within a given Scheme interpreter or client library. The tranport format is designed solely to allow both a compact and efficient read/print format and a common format for data transmission. The dtype specification provides for several different layers of protocols for dtype transport, each of an increasing level of complexity.

2.1.1 Binary Format

The dtype binary format defines a binary representation for dtypes. It is independent of transport mechanism, and used in all situations where no connection-level information is available. It is designed to be simple, reasonably compact, and easy to parse. All data is aligned on byte- (but not necessarily machine word-) level boundaries. The base protocol defines a small set of fundamental types that all dtype servers are required to understand. All more complicated types are required to be represented in terms of these fundamental types. This ensures that unknown data types will not cause a data stream to be unable to be parsed. If the dtype reader supports a given type, the object will appear unmodified to the reader. If not, the object will appear in the form of its simpler dtype representation.

Every dtype in the simple binary format is represented as the catenation of two parts: a type specifier, followed by a representation field as appropriate to the specified type.

2.1.1.1 The Type Field

The type field of a dtype is specified by a 64-bit object identifier that serves as a unique type identifier (and, in the dtype remote object system, refers to a class information object for the given type. See Section 2.3.1 [Remote Object System], page 23). As a space-saving measure, a dtype type field is reprented in binary format as a single byte referred to as a *prefix character*. If the value of the prefix character corresponds to that for the dtype class `typespec`, the prefix character and the subsequent 64-bit object identifier will be interpeted as a dtype of class

`typespec` specifying the type of the entire dtype. If the prefix character contains any other value, it will be interpreted as a complete specification of the given dtype type, and the object identifier will be omitted.

A subrange of the possible single-byte type fields is reserved by the dtype specification to refer to a predefined set of well-known types. Higher level network protocols may provide for ways to interactively allocate the remaining single-byte type fields for commonly used types on a connection-level basis. No position of privilege is implied by the existence of a single-byte type field for a given dtype type: such a field is only intended as an optimization for the most commonly used kinds of dtypes. The fact that some single-byte fields are pre-defined as allocated to certain 'well-known' types should be viewed as an optimization only.

2.1.1.2 Representation Formats

Following the type field of each dtype is the representation field. This field will contain an contain an integral number of bytes, and will be aligned on byte-boundaries. The format and interpretation of the representation data for a dtype are determined according to the type field for that dtype.

The representation field of a dtype can take any of the following formats:

`uint8` A single byte to be interpreted as an unsigned integer.

`uint16` Two bytes to interpreted as an unsigned integer in network (MSB-first) byte order.

`int32` Four bytes to be interpreted as an unsigned integer in network (MSB-first) byte order.

`uint32` Four bytes to be interpreted as a signed twos-complement integer in network (MSB-first) byte order.

`int64` Four bytes to be interpreted as an unsigned integer in network (MSB-first) byte order.

`uint64` Four bytes to be interpreted as a signed twos-complement integer in network (MSB-first) byte order.

`float32` Four bytes to be interpreted as an IEEE single-precision float in network (MSB-first) byte order.

`float64` Four bytes to be interpreted as an IEEE double-precision float in network (MSB-first) byte order.

dtype Contains a full dtype (both type and representation field) to be used as the representation of the type previously specified.

vector The format of a vector representation is:

[flags] <type> <flength> [length] <ltable> [elems]

, where [flags] is a single byte to be interpreted as a bitfield, <type> is an optional dtype type field to be used as the type for all of the elements in the vector, <flength> is an optional 16, 32-, or 64- bit unsigned integer, [length] is a 8-, 16-, 32-, or 64-bit unsigned integer, <ltable> is an optional table of 16-, 32-, or 64- bit unsigned byte offsets into the vector, and [elems] is the actual array data of the vector.

Bit zero of [flags] indicates if the array is heterogenous or homogenous. If 1, the vector will be a vector of dtype representations, to be interpreted according to the type specified by <type>. If 0, <type> will be omitted, and the vector will be a vector of compelete dtypes, each consisting of both a type and representation field.

Bits one and two of [flags] indicate the format to be used for the number of elements in the vector. If '0x0', the [length] field will be a <uint8>. If '0x1', [length] will be a <uint16>; if '0x2', [length] will be a <uint32>; if '0x3', [length] will be a <uint64>.

Bit three of [flags] indicates if the elements of the array are of fixed or variable length. If '0', each element of the vector will have fixed length as specified by <flength>. If '1', <flength> will be omitted. Each of the elements will then have variable length, and their lengths must either be parsed individually or derived from the specially-provided table <ltable>. If the vector is heterogenous, this bit will generally be unset, although execptional cases may exist (an array containing only objects of type <int32> and <float32>, for example). For homogenous arrays, the value of this field will generally be related to the type of object the array contains.

If bit three of [flags] is set, bits four and five indicate the format to be used for the entries in the (optional) offset table <ltable>. If bits four and five are '0x1', <ltable> will be an array of [length] 16-bit entries. If '0x2', <ltable> will contain 32-bit entries; if '0x3', <ltable> will contain 64-bit entries. If bits four and five are both zero, <ltable> will be omitted.

If bit three of [flags] is unset, bits four and five will represent the format used to interpret the length to be used for each element of the vec-

tor as specified by `<flength>`. If bits four and five are '0x1', `<flength` will be a `<uint16>`. If '0x2', `<flength>` will be a `<uint32>`; if '0x3', `<flength>` will be a `<uint64>`. If bits four and five are both zero, the array must be homogenous, and the field will be derived from the type specified by `<type>`. It is an error if either the array is not homogeneous or if the size of the entries cannot be derived reliably from the type field. If a value is provided both by the type for a homogeneous vector and by `<flength>`, the values for the two field sizes must be consistent.

Bit six of `[flags]` is reserved for future expansion and must be transmitted as zero. Bit seven of `[flags]` is reserved for type-specific interpretation by dtype types that make use of the vector representation.

If `<ltable>` is provided, each entry (of length specified by bits four and five of `[flags]`) will list the offset corresponding to the end of the corresponding element of the vector. The location given in `ltable[length - 1]` will therefore correspond to the end of the vector, and the length of vector element *n* can be found by examining `ltable[`*n*`] - ltable[`*n* `- 1]`.

In some circumstances, a dtype type field will pre-define the format or type used by a given vector. In such cases, one or several of the mandatory or optional fields of a vector representation may be omitted and assumed as already specified. Such modifications of the vector format will be made explicit when they are used.

In the special case where bit one of `[flags]` is '1' (the vector is homogeneous), the value of `<type>` is `bool`, and both bits four and five of `[flags]` are zero (the representation format is to be derived), the boolean array will be sent in packed format — i.e., eight boolean values per byte, shifted left and padded to the nearest byte. To send an array of boolean in unpacked format, `<flength>` can be specified explicitly as `1`.

2.1.1.3 Fundamental Types

Each dtype has a representation field that is interpreted according to the format specified by the type field. Some dtype types, known as "fundamental types," are required to be understood by all dtype implementations, and can use any of the predefined data formats as their representation. Sixty-four entries of the 8-bit type namespace and a 32-bit segment of the 64-bit type namespace are reserved for fundamental dtype types.

Non-fundamental dtypes must have representations composed of format `<dtype>`. This is to ensure that implementations that do not support a given type of dtype will at least be able to parse and understand it to a limited degree. Two-way dtype protocols may negotiate alternate transmission schemes for non-fundamental dtypes. The presence or absence of a eight-bit type code for a dtype has no effect on whether or not a given dtype type is considered fundamental.

The following fundamental types are required by the dtype protocol:

`null` Representation field is empty. Corresponds to the scheme object `#null`.

`bool` Representation field is a `<uint8>`. If '0x0', corresponds to the scheme object `#f`. Otherwise, corresponds to the scheme object `#t`.

`true, false`
: Representation fields are empty. Correspond to the Scheme objects `#t` and `#f`, respectively.

`int32, int64, uint32, uint64, float32, float64`
: Representation field is a `<int32>`, `<int64>`, `<uint32>`, `<uint64>`, `<float32>`, or `<float64>`, accordingly. Corresponds to the scheme integer or floating-point number of the appropriate size and type.

`char` Representation is a `<uint8>` to be interpreted as an 8-bit character.

`unichar` Representation is a `<uint32>` to be interpreted as a Unicode character.

`oid` Representation is a `<uint64>` representing a global object identifier for distributed database support. See Section 2.3.1 [Remote Object System], page 23 for more information.

`type` Representation is the same as `oid`, except that the specified `oid` is declared to represent a `dtype` type.

`vector` See the format entry for `vector`.

`pair` Representation is the same as that of `vector`, except that the `[length]` field need not be specified and will always be '2'. Bits one and two of the `[flags]` field of the vector must both be zero and will be ignored.

`packet` Representation is a vector of `uint8` to be interpreted as raw data string. The `<type>` field of the vector is omitted and specified as `char`. Bits one, four and five of the `[flags]` field must all be zero and will be ignored.

`string` Representation is a vector of `char` to be interpreted as an 8-bit string. Like in `packet`, the `<type>` field of the vector is omitted and specified

as `char`. Bits one, four and five of the `[flags]` field must all be zero and will be ignored.

`unistring`
: Representation is a vector of `unichar` to be interpreted as an 32-bit Unicode string. Like in `string`, the `<type>` field of the vector is omitted and specified as `unichar`. Bits one, four and five of the `[flags]` field must all be zero and will be ignored.

In the absence of a connection-level agreement to the contrary, all other data types must be represented in terms of these fundamental types. If the type specifier for a dtype does not fall within the predefined range of (64 entries of the 8-bit type namespace; 2^{32} entries of the 64-bit type namespace) it must use a representation field of format `<dtype>`. Such a format will be referred to as a *compound representation*. The requirement that all non-fundamental dtypes be represented in terms of fundamental dtypes is not as expensive as it might at first seem. Assuming that a single-byte type code is allocated for the new type, using a compound for a new dtype type requires only an extra byte of overhead per dtype for short objects (for the new type field). For longer fixed-size objects, a `vector` representation can be used. In neither case is the additional overhead greater than the minimum of one byte or 15% of the data being represented.

In interactive or semi-interactive communication environments, alternate communication formats for compound dtype types may be negotiated.

2.1.2 Interactive Protocols

2.1.2.1 Connection Protocol

The dtype connection protocol is used for network servers or in other situations where connection-level state information can be maintained. Regardless of the actual relationship between the two communicating processes (which might in fact be peer-peer, server-client, or client-server), I will refer to the process which initiates the connection to as the client, and to the process which accepts the connection as the server.

The dtype connection protocol defines the new type `connection-parameter`. An object of type `connection-parameter` can contain an arbitrary dtype as its representation. Typically, this dtype will be a single sequence to be interpreted in expression-like format.

The following connection parameters are currently supported:

`(connection-version` *version*`)`
: Sent from the server to the client upon initial connect. Specifies the version of the dtype connection protocol to be used for the current connection. The current value for *version* should be `''original'`.

`(symbol-intern` *symbol value*`)`
: Can be sent from either party to the other. Specifies that for the duration of the lifetime of the connection, the value of *symbol* can be specified in shorthand by the `int32` *value*. Symbols must still be preceded by a type code as before: it is only the representation field which changes. A separate intern table is maintained for each direction of communication.

`(typecode-intern` *typecode tp*`)`
: Can be sent from either party to the other. Specifies that for the duration of the lifetime of the connection, the `uint8` *typecode* will refer to the `typespec` specified by *tp*. The value of *tp* must be within the range reserved for connection-defined typecode allocation. A separate typecode translation table is maintained for each direction of communication.

`(type-information` *tp*`)`
: Can be sent from either party to the other. Provides the recipient with meta-information for the class specified by the `type-information` *tp*. Such an object should be sent at least once before each non-well-known type that is sent over a dtype connection. If such a structure is not sent, the type of the object may or may not be recognized on the receiving client.

2.1.2.2 Command Protocol

The dtype notification protocol is a convention layered on top of the dtype connection protocol. In this protocol, clients send dtype expressions to the remote server. Upon receiving a dtype from a client, the server evaluates that dtype in an environment specific to that client, and returns the evaluated result back to the client as a dtype.

Note that although the overall protocol structure is that of alternating queries and responses, the existence of `connection-parameter` objects may cause the actual dtype traffic to not follow an alternating pattern.

The following two additional types are used to represent meta-information about dtype command servers:

`server-location`
: Inherits from `map`. A `server-location` object has three required fields:

  `host`
  : Contains a `dt_string` that specifies the hostname on which the service is located.

  `port`
  : Contains a `int32` specifying the port number on *host* on which the command port for the specified service can be found.

  `notification-port`
  : Contains a `dt_int` specifying the portnumber on *host* on which the notification port for the specified service can be found.

`server-description`
: Inherits from `map`. A `server-description` object has two required fields:

  `commands`
  : Contains a `map` mapping symbol names to descriptions of their function.

  `notifications`
  : Contains a `map` mapping symbol names to descriptions of the event of which they are used to notify clients.

All dtype command servers must support the following commands:

`(server-description)`
: Returns a dtype of type `server-description` describing the current server.

2.1.2.3 Notification Protocol

The dtype command protocol is layered on top of the dtype connection protocol, and is used to provide asynchronous notification of events to connected clients. In this protocol, clients are not permitted to send data to the remote server. Servers may at any time send dtype data to any connected client, which should be interpreted as notification of the event specified by that dtype.

2.1.2.4 Example: Garden CD Server

As an example, the CD player in the Terminal Garden of the MIT Media Laboratory is controlled by a dtype server running on the machine `bluevelvet.media.mit.edu`.

The command port of the CD player is on port `21397`, and supports all the R^4RS Scheme functions, as well as the following commands:

`(cd-play-track` *track*`)`
: Causes the CD player to play the track specified by the integer *track*. Returns '`#f`'.

`(cd-eject)`
: Causes the CD player to eject the currently loaded CD. Returns '`#f`'.

`(cd-pause)`, `(cd-resume)`, `(cd-stop)`
: Causes the CD player to pause play, resume play, or stop completely, as approprate. Returns '`#f`'.

`(cd-status)`
: Returns the status of the CD player in the form of a dtype `map`.

The notification port of the CD player is on port `21398`, and provides the following notifications:

`(insert)`, `(insert` *id*`)`
: Indicates that a CD has been inserted into the CD player. The string *id*, if present, specifies the registered unique identifier for that CD. That identifier can be looked up in a global database (currently provides as a separate dtype server) to find track, title, ant artist information about that CD.

`(eject)`
: Indicates that the current CD has been ejected.

`(pause)`, `(resume)`, `(stop)`
: Indicates that the CD player has paused, resumed playing, or stopped playing, as appropriate.

`(play)`
: Indicates that the CD player has begun playing.

`(track` *t*`)`
: Indicates that the CD player is now playing track *t*.

2.1.3 The Dtype Object System

The dtype object system provides a simple object system for the dtype library similar to that of TinyCLOS. Unlike in most Scheme implementations, the dtype

object system is integral both to the semantics and the implementation of the dtype type system. Although arguably at odds with the minimalist design of the Scheme specification, such a design simplifies the specification and implementation of remote dtype objects.

`object` All dtype objects are subclasses of the type `object`. The `object` class supports the following methods:

- `type` Returns a `typespec` representing the type of the object.
- `put` *nrep* Sets the value of the dtype to correspond to the representation value *nrep*. The argument *nrep* must consist solely of fundamental dtypes.
- `get` Returns a representation for the internal value of the dtype, which must be expressed solely in terms of fundamental dtypes.

`bag` The `bag` class inherits both its representation and its methods from `object`. Corresponds to a set of objects whose (possibly non-unique) elements can be iterated in a non-deterministic fashion. A `bag` is represented in terms of fundamental dtypes as a `vector` containing the contents of the `bag` in arbitrary order. The `bag` class adds no methods to `object`.

`sequence` The `sequence` class inherits both its representation and its methods from `bag`. Corresponds to a set of objects whose (possibly non-unique) elements can be iterated in a deterministic fashion. A `sequence` is represented in terms of fundamental dtypes as a `vector` containing the contents of the `bag` in sequential order.

The `sequence` class adds the following methods to `bag`:

- `element` *n* Returns the *n*th element of the sequence.
- `set-element` *n dt*
  Sets the nth element of the sequence to *dt*.
- `insert` *n dt*
  Inserts the dtype *dt* at position *n* in the sequence.
- `append` *dt* Appends the dtype *dt* to the end of the sequence.

`set` The `set` class inherits both its representation and its methods from `bag`. It behaves like `bag`, except that no object can be present in a set more than once. The test used for equality is semantic equivalence

(in Scheme, this corresponds to `equal?`, not `eq?` or `eqv?`). A `set` is represented in terms of fundamental dtypes as a `vector` containing the contents of the `set` in sequential order.

The `set` class adds the following methods to `bag`:

`lookup` Returns the *n*th element of the sequence.

`insert` *dt* Inserts the dtype *dt* into the set.

`contains?` *dt*
: Returns '`#t`' if the set contains the dtype *dt*; '`#f`' if it does not.

`map` The `map` class inherits both its representation and its methods from `bag`. It is used to manage a set of key-value association pairs, all of which have a unique key. The test used for key equality pis semantic equivalence (in Scheme, this corresponds to `equal?`, not `eq?` or `eqv?`). A `map` is represented in terms of fundamental dtypes as a `vector` of `pair` listing each key-value pair in sequential order sorted by key.

The `map` class adds the following methods to `bag`:

`lookup` *key*
: Returns the value associated with *key* in the map. Returns an exception if *key* is not contained within the map.

`insert` *key val*
: Associates the value *key* with *val* in the map. If *key* is already contained in the map, replaces the previous value associated with *key* with *val*.

`contains?` *dt*
: Returns '`#t`' if the map contains a value for *key*; '`#f`' if it does not.

`environment`
: The `environment` class inherits its representation from `sequence` and its methods from `map`. It is used to represent Scheme-style environment structures. A Scheme environment is represented as a sequence of maps, where each map in the sequence shadows the entries in the maps that follow it.

The `environment` class adds the following methods to `map`:

`lookup` *key*
: The same as in `map`, except that each map in the sequence is examined in turn for *key*. As soon as a value for *key* is

found, `lookup` returns that value. If no value is found, `map` returns an error.

`define` *key value*
: Binds *key* to *value* in the first map of the environment. If *key* is already bound in that map, replaces the existing binding.

`set` *key value*
: Changes the binding of *key* to *value* in the first map in which *key* is bound to any value. Returns an error if *key* is not bound to a value in any of the maps in the environment.

`type-information`
: The `type-information` inherits both its representation and its methods from `map`. It is used to provide type information about a dtype type. The map contained in a `type-information` object contains the following fields:

`extends`
: Contains a `typespec` representing the dtype class extended by the type being described. Since all user-defined types are required to be able to represented in terms of fundamental dtypes, all user-defined dtypes will types will at the very least be extensions of the class `object`.

`inherits`
: Specifies base classes from which the dtype class inherits methods, in left-to-right order.

`slots`
: Contains a `map` specifying a binding for each of the slot entries provided specifically by the class being defined. Each entry in the map will associate a `symbol` (the slot name) to a slot description structure. The slot description structure will itself be a map containing the following fields:

`name`
: A `symbol` representing the name of the slot entry.

`description`
: A short description of the purpose of the slot entry.

`implementation`
: A Scheme function which, when called with an instance of the object representation and optional arguments, performs the specified opera-

tion upon the object. The mechanism to specify the environment required by this function is not yet defined.

`number`
: The `number` class inherits both its representation and its methods from `object`. It is used to represent arbitrary numeric data. The `object` class adds no fields to `object`.

`uinteger`
: The `integer` class inherits its representation from `packet` and its methods from `number`. It is used to represent unsigned integers of arbitrary magnitude. The packet should contain the magnitude of the integer in MSB-first format and padded to the left. The `uinteger` class adds no fields to `object`.

`integer`
: The `integer` class inherits its representation from `pair` and its methods from `number`. It is used to represent signed integers of arbitrary magnitude. The `car` of the pair should be a single `bool` representing the sign; the `cdr` should be a `uinteger` representing the magnitude. The `integer` class adds no fields to `object`.

`rational`
: The `rational` class inherits its representation from `pair` and its methods from `number`. It is used to represent rational numbers. The `car` of the pair should be a `integer` representing the numerator; the `cdr` a `uinteger` representing the denominator. The `rational` class adds the following fields to `pair`:

  `numerator`
  : Returns the numerator of the rational number.

  `denominator`
  : Returns the denominator of the rational number.

`symbol`
: The `symbol` class inherits its methods and its representation from `string`. A `symbol` is the same as a `string`, except the string value is to be interned and interpreted as a Scheme symbol in a way equivalent to the behavior of the Scheme function `string->symbol`.

`unisymbol`
: The `unisymbol` class inherits its methods and its representation from `unistring`. A `unisymbol` is the same as a `unistring`, except the string value is to be interned and interpreted as a Scheme symbol in a way equivalent to the behavior of the Scheme function `string->symbol`.

`error`
: The `error` class inherits both its representation and its methods from `object`. It is used to represent the existence of an error in processing

or an unexpected condition. The `error` class can be thought of as a tag that can be attached to an arbitrary dtype to indicate that it represents an error value. The `error` class adds the following fields to `object`:

`value` Returns the object corresponding to the value of the error.

2.1.4 Conclusion

Using dtypes as a data representation allows one to store and transmit arbitrary structure portably and safely. Used within a single program, dtypes eliminate the need to define, understand, or parse custom configuration or data file formats. By using the dtype (either ASCII or binary) protocol as the format for a configuration file, arbitrary structure can be represented and automatically parsed without having to write any extra code.

Used as a communications layer, the dtype protocol allows clients to efficiently transmit arbitrary structured data over communications links using a common and easily parsed format. Systems such as XDR, ASN.1, and MiG require the data format used to be defined in advance and understood by both parties to the communication. A partial knowledge of the data being transmitted is of no use: both the sender and receiver must be using a data format description compiled from the same source.

By contrast, using the dtype library for a network communications layer allows systems with only a partial knowledge of the data format being used to gain at least a partial understanding of the data being transmitted. In addition, the dtype meta-object protocol allows communicating processes to define new types and exchange type information interactively, without compromising the ability of less sophisticated clients to be able to at least partially interpret the data.

2.2 Dsys and Derived Iostreams

The *dsys* and *derived iostream* libraries provide a simple and powerful interface to common networking facilities. The `dsys` library makes TCP/IP connections through the C++ `iostream` facility. The *derived iostream* library allows the programmer to attach filtering functions to both the head and the tail of C++ iostreams. These functions can be used to transparently add compression, encryption, or other filters to arbitrary stream connections. Finally, the `dsys_pollblock` and `dsys_server` classes provide a mechanism to easily implement network servers and perform event-handling services in a general format. When combined with the dtype protocols, these services provide a simple way to write extensible network servers.

2.2.1 Dsys

The dsys library provides a binding between TCP/IP networking services and the C++ iostream facility.[1] In the dsys library, a dsys_connection represents a single TCP/IP connection to a given host and port number. The primary interface to a dsys_connection is through an istream and ostream made accessible through member functions. Everything written to the ostream is sent to the network connection; everything received from the network connection is made available on the corresponding istream.[2]

2.2.2 Derived Iostreams

The *derived iostream* library makes it possible to insert custom filters into either end of an iostream. A iostream filter is a single C++ function that takes three arguments: a reference to an istream from which raw data should be read, a reference to an ostream to which filtered data should be written, and a closure specific to each iostream that can be used to store stream-specific state information. As a a special case of the derived iostream library, the *derived pipestream* allows one to use an external program as a filtering function.[3]

---

[1] The iostream facility of C++ replaces the FILE * used by the standard C library. The standard C++ library provides for a number of iostream formats: the strstream classes operate on buffers of raw memory; cin, cout, and cerr provide the standard input/output/error facilities, and the fstream classes provide an interface between iostreams and disk files. Using the polymorphism features provided by C++ library interfaces can be written to use generic iostreams, with different things happening to the input/output depending on the characteristics of the stream passed in.

[2] It would also have been possible to use a single iostream to represent both incoming and outgoing data, but this class seems to have been removed from more recent versions of the standard C++ library and is not available in all implementations.

[3] The implementation of a derived pipestream would normally be a trivial application of the derived iostream library, except for the need to avoid deadlock between the reading and writing portions of the filter.

2.2.3 Dsys Pollblocks / Servers

Finally, the `dsys_pollblock` and `dsys_server` classes provide an event-driven mechanism to manage multiple network connections. The `dsys_pollblock` class provides a generic interface to the UNIX-inspired `select()` interface; the `dsys_server` class provides a convenient mechanism to create multi-client network servers.

2.2.4 Conclusion

Used alone, the `dsys` and derived `iostream` libraries enable a number of useful network applications. Programs that use network I/O can now be written to use generic stream interfaces (as with `inetd`); a `dsys_connection` converts these generic interfaces to actual network transactions. By attaching filters to input and output streams, one can change the characteristics of an `iostream` without modifying the associated client application. A compression scheme can be added to a network protocol simply by attaching a compression filter to both ends of the `dsys_connection`; a secure compressed connection can be implemented as a compression filter followed by an encryption function that uses the `iostream`'s closure to store the key.

2.3 The Dtype Distributed Computation Environment

The dtype transport protocol provides a simple and expressive mechanism for transporting structured data in the absence of predefined formats. Accordingly, the `dtype` transport protocol makes a good basis for a distributed computation environment, in which arbitrary clients and servers can communicate by exchanging dtype structures.

2.3.1 Remote Object System

The dtype remote object protocol provides a uniform way to create, manage, and invoke opertaions on remote Scheme objects.

The remote object namespace uses a 64-bit unsigned integer as a global network object identifier. Segments of this namespace can be delegated to subordinate authorities on arbitrary subregions (often, but not necessarily related to bit- or byte-boundaries of the namespace).[4] A bit range, once delegated to a sub-authority, is

[4] Despite possible appearances otherwise, the ability to delegate namespace regions on arbitrary regions should have little effect on the efficiency of nameserver

entirely the responsibility of that sub-authority, which may sub-delegate arbitrary regions as it finds appropriate.

The master dtype object namespace server resides on the host `oid-resolver.services.dtype.org` on port `23889`. This host entry will have multiple IP addresses for redundancy and load-balancing as appropriate. Private caching secondary servers will also likely be maintained at many locations.

A dtype object server will use the dtype command protocol and must support the following function:

`(oid-resolve` *oid*`)`

> If *oid* is managed by the given object server, returns `#t`. Returning a value of `#t` to `oid-resolve` implies a willingness to support remote access to that object. If *oid* is managed by a delegate of the given object server, returns a `sequence`, where each entry is a `map` which contains at least the following entries:
>
> `host` — Contains the hostname of an object server responsible for the object.
>
> `port` — Contains the port number associated with that object server.
>
> `flags` — Contains a `sequence` of `symbol` specifying flags appropriate to the object server. The only flag currently supported is `'read-only`, which indicates that the specified object server will only allow accesses to the object in a read-only fashion.

In addition, if a dtype object server supports remote access to any object, it will provide the following function:

`(oid-invoke` *oid method* `.` *method-args*`)`

> Calls the method specified by the symbol `method` on `oid` with the arguments specified by *method-args*. Returns an `error` if the object is not managed by the given object server, if the method is not supported for the particular object type, or if any other error occurs.

---

implementations. One simple way to keep track of which objects belong to which namespace subregions is to keep the lower bound of each range in a sorted array. To resolve an object identifier, binary search to find the largest lower range bound that is still less than the object identifier. Then check to see if the object identifier is within the range for that lower bound. If so, delegate to the entry for that range. If not, then reject the identifier as invalid or (in the case of a caching secondary server) refer to the primary authority.

2.3.2 Meta-Object Protocol

The dtype meta-object protocol is designed to allow dtype applications to communcate information about user-defined types in a portable format.

The dtype meta-object protocol works by exchaging objects of the well-known type `type-information`. See Section 2.1.3 [The Dtype Object System], page 16 for more information on the class `type-information`.

In particular, the `dtype` class library (as distinct from the `dtype` transport protocol) provides a way to allow remote dtype objects to behave pseudo-transparently as if they were objects on the local machine. See Section 2.4.2 [The Dtype Method System], page 27 for more detail.

2.4 The Dtype Class Library

As a mechanism for inter-process and inter-machine communication, the dtype transport protocol provides a good way to represent, store, and transmit arbitrary Scheme-style data structures. Interfaces to the dtype protocol have been written for a number of widely-available Scheme implementations; among them are `scheme48` and `GWM`.

This section will discuss the design and implementation of the dtype class library, a C++ class library that provides access to Scheme data structures from C++. The dtype class library provides a mechanism to create, manipulate, and evaluate Scheme data structures from C++. It is maximally space-efficient, and time-efficient where possible without compromising the cleanliness of its interface.

As a C++-accessible class library that uses the `dtype` transport protocol, the `dtype` class library, when combined with the `dsys` networking libary, provides a simple way to create flexible network servers and protocols. The `dtype` class library should not, however, be confused with the `dtype` transport protocol; the fact that they share a common name is a historical accident only. The `dtype` class library is only one of a number of packages that can speak the `dtype` protocol directly.

2.4.1 Data Structures

The `dtype` library provides a simple and space-efficient way to create, manipulate, and evaluate Scheme-like data structures from C++. The fundamental component of the `dtype` library is the *dtype*. A dtype represents a single Scheme object of arbitrary type. All of the fundamental Scheme types are available (integers, strings, symbols, pairs, vectors, etc.), as well as a number of other types not strictly required

by the Scheme standard (sets, queues, associative arrays, etc.). Several domain-specific type libraries are available (to support image processing functions, X server connections, etc.), and new types can be added either by application programs or at runtime by dynamically loading object files.

Although there is a one-to-one mapping between each C++ dtype type structure and a transport layer `dtype` representation, the two objects—even though they share the same name—must be viewed as conceptually different. For the rest of this section, I will use the word 'dtype' to refer to the C++ class structure, not the associated transport layer representation.

The objects used by the `dtype` library are designed to require as little memory as possible. Each dtype is represented by a short value (32 bits in all current implementations) called an *immediate value*. The initial bits of the immediate value contain a variable-length field used to determine the type of the object in Huffman-code like fashion. The remaining bits store the actual data value of the object.

The use of a variable-length type field allows dtype classes to maximize the usage of the available address space by allocating more bits of data to the more commonly used types. For example, in current implementations, a `dt_int` contains two bits of type data field and thirty bits of signed integer data. A `dt_symbol` contains four bits of type information and twenty-eight bits of data (for a total of 2^{28} possible symbols). A `dt_bool`, by contrast, contains thirty-one bits of type information and a single bit for the data value. This implies that the `dt_int` class uses one fourth of the available address space for dtypes in memory, whereas the `dt_bool` class uses only a very small fraction.

Some `Scheme` objects are too large to fit in an immediate value. The `dt_pair` class, for example, must at a minimum store the two dtypes that make up the `car` and the `cdr` of the pair. The `dt_pair` class uses four bits of type information and has as a data value a twenty-eight bit index into a garbage-collected heap of 64-bit objects (the first 32 bits of each object represent the `car` of the pair; the last 32 bits represent the `cdr`). This allows a maximum of 2^{28} pairs to be represented by the `dtype` library. Although the use of a separate heap for each class type can be a burden to implementations seeking to maximize locality of reference, the use of a separate heap for each type increases the available address space for objects and makes it easier to manage the storage allocator for each type (especially those types that contain only objects of fixed size).

It is not necessarily desirable to have a separate Huffman-based typecode for each type used by the dtype class library. Since the allocation of a typecode reserves the

entire addres space below that typecode for the specified type, the allocation of too many distinct typecodes will fragment the available address space, and make it more likely that an implementation will find itself unable to allocate a new object of any given type. To avoid this problem, the dtype library allows less-commonly used structures to be stored using a *compound representation.* A `dt_compound` uses the same format as `dt_pair`, except using a different code. Unlike `dt_pair`, the `car` of a `dt_compound` must be an object of type `dt_type`, which corresponds to a dtype type information structure. The `cdr` of the `dt_compound` will be the representation of the `dtype` to be used for the new type. Such an organization allows the less commonly used dtype structures to share a single namespace, while still providing an efficient way to determine the type of a given object.

The ability to variably allocate bits to type and data notwithstanding, the choice of a 32-bit format for all dtype representations in memory provides a strict upper bound of 2^{32} possibly distinct objects in any particular program. This becomes limiting only in the face of machines with execptionally large amounts of both virtual and physical memory, at which point it becomes reasonable to require the use of 64 bits for each immediate value instead of 32.

In Scheme, the type of an object is associated with the object itself, not with the variable name (as in C++, Clu, etc.). Accordingly, all objects in the dtype library inherit from the base class `dtype`: any dtype can be assigned to any other dtype, and data structure classes (such as sets and associative arrays) can contain dtypes of any type as any of their elements. Compile-time type checking is used wherever possible by the library; run-time type errors are handled by the dtype exception system defined later.

2.4.2 The Dtype Method System

To further conserve space while still providing for polymorphism, the `dtype` library provides for a simple CLOS-style method system based on dispatch off the type field. For example, all dtypes have virtual functions to read and write their contents in both ASCII and binary formats, and all sequence types (lists, vectors, sets, etc.) contain routines to iterate through their members. These functions are indistinguishable from standard C++ virtual functions, but they require no additional memory. For comparison, the naive implementation for a `dt_int` would require 128 bits on an DEC Alpha (64 bits for the virtual function pointer, 32 bits for the integer value, and the other 32 bits lost to alignment requirements). A `dt_int` requires only 32 bits for the entire object.

The C++ implementation of the dtype method system works by maintaining a separate class hierarchy with a separate class for each different subclass of dtype. Each entry in this class hierarchy contains a table that maps slot names to functions that implement that slot's behavior. The use of the C++ inheritance mechanism to construct the slot table ensures that equivalent slots will have identical locations in all of the subclasses of a given dtype class.

Such a design makes it possible to transparently support remote dtype objects within a dtype interpreter. When the object identifier for remote dtype is received, the receiving interpreter uses the metaclass information for that dtype to build a dispatch table that refers each method invocation to the remote server for evaluation. This allows dtype clients to operate upon dtype objects at remote servers using exactly the same semantics as if the object were local, without severly compromising the cost of access for local objects.

2.5 The Dtype Scheme Evaluator

Viewed solely as a C++ class library, the `dtype` class library provides an extensive set of features: a rich and user-extensible set of polymorphic data types, a robust run-time type checking and extension handling facility, and a common read/write syntax for easy network transportability. As a Scheme interpreter, the `dtype` class library includes provisions for the dynamic loading of both new primitive types and compiled C++ functions and for exact application control of evaluator actions, including environments for the safe execution of foreign applications and a continuation-based multi-threading and exception-handling system.

The `dtype` library is not, however, designed, to be a production-quality `Scheme` programming environment. In particular, it does not provide a byte-compiler or microcode-based interpreter. The `dtype` library uses the `C++` stack for function calls, and many of the more advanced `Scheme` features are provided in naive and often inefficient ways. It is assumed that the `Scheme` interpreter will be used primarily for higher-level scripting functions. Since the interface between `Scheme` and `C++` is so clean, I expect that most low-level functions will be implemented directly in `C++` for efficiency.

2.5.1 Hybrid Programming

The `dtype` library has a number of features to support its use as a hybrid programming language. The `dtype` library allows dynamic loading and execution of

new types, new functions (written either in Scheme or in C++), and arbitrary data structures.

The utility of `dtype` as a C++ class library is also its primary strength as a Scheme interpreter. Since the data structures and functions defined and used by the `Scheme` interpreter of `dtype` are immediately accessible through the provided `C++` class library, it is possible to use `dtype` as a hybrid programming language, using both `C++` and Scheme in the same program without the inconvenience of typical foreign function interfaces. Since arbitrary C++ functions and types can be loaded at runtime, `dtype` can be used as an extensible scripting language where low-level functions are written in C++ for maximal efficiency and the Scheme interpreter is used for rapid prototyping, garbage collection, safety, and higher-level functions.[5]

Since the capabilities of a `dtype` script are limitied by the Scheme environment in which it is evaluated, it is possible to create a safe execution environment for `dtype` programs. Because of the linkage between the Scheme interpreter and the C++ class library, `dtype` scripts can easily be translated into C++ and compiled while remaining safe to execute. In addition to a restricted evaluation environment, `dtype` functions can be limited in their usage of processor resources, memory, and network bandwidth. When the evaluation of an expression violates a resource limit, the expression will raise an exception containing an explanation of the resource violation and a continuation[6] to permit continued evaluation.

For increased safety, the `dtype` library integrates the C++ exception handling facility into the Scheme interpreter. Exceptions raised by `dtype` primitives (if not explicitly handled by the C++ function) are automatically trapped by the evaluator and converted into Scheme exceptions. This allows C++ primitives to have automatic error handling performed by all `dtype` routines without the need for explicit error checking.

3 Applications to Multimedia

[5] Hence the emphasis of space conservation over speed in the evaluator: Scheme functions that are too slow can be re-written easily in C++, but the memory usage of a data structure will remain constant regardless of which language is used.

[6] A generalization of `C's setjmp()/longjmp()` facility, a continuation is a snapshot of a programs execution state to which it is possible to return by evaluating the contiunuation as a function call.

The hybrid nature of the `dtype` library makes it particularly useful for creating multimedia applications. Multimedia processing requires real-time processing of large quantities of data, as well as low-level programming to support direct access to specific hardware. Accordingly, any useful multimedia system will depend upon C, C++ or some other low-level systems programming language. The use of the `dtype` library allows one to gain the benefits of a highly extensible Scheme environment without either losing efficiency or sacrificing the clarity of one's C++ environment. In this section, I will describe the use of the `dtype` library to implement the *Media Bank*, a distributed interactive multimedia environment.

The implementation of the presentation layer of the Media Bank depends on three layers of object hierarchies, listed in decreasing order of implementation-dependence. The first, the object layer, defines the format of media bank objects the data contains the Media Bank items that contain the semantic information of the object being represented. The second, the control layer, is based on a collection of Scheme objects that manage Media Bank data. The organization of the objects in this layer is close, but not identical, to that of the object tree. The third, the display layer, is based on the X Toolkit, and controls the actual display of the Media Bank object as well as the interaction between Media Bank objects and the program in which they are being used.

3.1 Object Layer

Every media bank object is repesented by a single `map` of keyword-value pairs.

The list of possible media bank objects is organized into a simple singly-inheriting hierarchy of types that is independent of the dtype type system (although this may change in the near future). The type of a media bank object is specified by a sequence of symbols specifying an inheritance hierarchy in left-to-right order.

The following object types are currently supported:

`'(item image mjpg)`

Represents a sequence of video in MJPG format. Contains the following fields:

`type` Must be `'(item image mjpg)`.

`data` Contains a string with a URL referring to the video data.

`frames` If present, contains the number of frames in the video segment.

`fps` If present, specifies the natural frame rate of the video segment.

`duration` If present, specifies the natural duration of the video segment.

If `frames` is specified, it must be consistent with the number of frames actually present in the video data. If both `fps` and `duration` are specified, they must be mutually consistent.

`'(item audio raw)`
Represents a segment of raw audio data. Contains the following fields:

`type` Must be `'(item audio raw)`

`data` Contains a string with a URL referring to the audio data.

`duration` If present, contains the natural duration (as a rational number, in seconds) of the video segment.

`encoding` Must be `'twos-complement`.

`'(item application subrange)`
Represents a sub-region of a previously defined media bank object. Contains the following fields:

`type` Must be `'(item application subrange)`.

`object` The name of the object into which the subrange refers. Should be a string containing the name of a media bank object.

`offset` The offset into the object at which the subrange begins.

`duration` The duration of the subrange within the object. If the duration specified for a subrange would cause the subrange to go past the end of the object it references, the remainder of the subrange will be assumed to apply to a null object.

`'(item application playlist)`
Represents a script that should be used to assemble a multimedia presentation from multiple sources.

Contains the following fields:

`version` Contains the version of the media bank script specification for which the give playlist is written.

`script` Contains the body of the script, specified as a *command list*. A command list consists of a sequence of commands which are executed in sequence by the script player.

The media bank script format supports the following commands within a command list:

`(wait` *n*`)` Causes the current evaluation thread to pause for *n* seconds. Command interpretation will resume as soon as *n* seconds have elapsed.

`(play` *ospec*`)`
Causes the current evaluation thread to begin displaying the presentation specified by the *play specifer ospec*. Command interpretation will resume as soon as the object playback specified by *ospec* has completed.

`(play-multiple` *svec*`)`
Each element of the sequence *svec* must be a command list. Causes an evaluation thread to be created for each element in *svec*. Each of the evaluation threads will be executed concurrently. Command interpretation for the current evaluation thread will resume when the execution of all of the command lists in *svec* has completed.

A play specifier is represented as a map containing the following fields:

`object` Specifies the name of the object to be played.

`x`

`y` Specifies an `x` or `y` offset relative to the parent to be used for any graphical operations performed by the object being played.

`xscale`

`yscale` Specifies an x- or y- scaling factor to be applied to any graphical operations performed by the object being played.

`duration` Specifies that the duration of the object should be scaled in an appropriate manner from the natural duration to the duration specified.

`tscale` Specifies a scaling factor to be applied to the natural duration of the object.

`'(item application sprog)`

Represents a Scheme application that should be executed on the displaying client. An object of this type should contain the single field `function`. This field should map to a lambda expression that creates a single hierarchy of molecules to be executed by the client. Since an arbitrary Scheme program can be included in the event handling routines, this existence of this type allows interactive applications to be sent from media bank servers to media bank clients. Based on the identity of the server sending the embedded application, the function received from the remote server may or may not be executed in a restricted environment.

For example, the object tree for the playback of 'Terminator II' contains the following items:

`<t2-movie>`

A playlist requesting that the objects `<t2-video>` and `<t2-audio>` be played concurrently.

`<t2-video>`

A playlist requesting that the video objects `<t2-video-ch1>` through `<t2-video-ch78>` be played in sequence.

`<t2-audio>`

A playlist requesting that the video objects `<t2-audio-ch1>` through `<t2-audio-ch78>` be played in sequence.

`<t2-video-ch1> ... <t2-video-ch78>`

The individual video objects for each chapter of 'Terminator II'.

`<t2-audio-ch1> ... <t2-audio-ch78>`

The individual video objects for each chapter of 'Terminator II'.

3.2 Control Layer

The control structure of the Media Bank playback client is controlled by a collection of Scheme objects which I will call 'molecules'. Each molecule used by a Media Bank client will represent a semantic element of the object being displayed. Individual molecules can be created, destroyed, and reconfigured rapidly and continuously throughout the playback of an object.

Molecules share a simple object protocol based on the Scheme object system. The following object methods are supported for all currently defined molecules:

`(create)` Creates the molecule and all its associated components. In an Xt-based video molecule, this will include managing and mapping all of the Xt widgets associated with the molecule. In the case of an audio molecule, this would include creating the audio context used to communicate with the audio server.

`(destroy)`
Destroys the molecule and all its associated components. In the case of an Xt-based video molecule, this includes destroying all of the Xt widgets associated with the molecule. In the case of an audio molecule, this would include destroying the audio context used to communicate with the audio server.

`(update` *timecode*`)`
Updates the molecule to be current to the timecode specified by *timecode*. In the case of an video object, this will involve displaying the appropriate frame on the screen. In the case of an audio object, this will involve buffering the appropriate amount of audio. For composite object types like playlists, this will mean sending the appropriate `(update)` messages to child molecules.

`(duration)`
Returns the duration of the item if played as a multimedia object, in seconds.

3.3 Display Layer

The low-level interface between the format layer of the Media Bank and the display hardware is based on the X Toolkit. A set of higher-level X Toolkit widgets is also provided to allow higher-level programs to use media bank applications as embedded data types.

The following higher-level widgets are currently provided:

3.3.1 MBPlayer

When given the name of a Media Bank item, the MBPlayer widget creates the molecules appropriate to that item as children of itself. It then provides the top-level application with a simple multimedia browser that sends `(update)` messages to the molecule it contains.

In addition to being able to synchronously synthesize multimedia presentations from media bank objects of the appropriate types, the MBPlayer widget can also evaluate arbitrary portions of `Scheme` code (sent by an object of type `'(item application sprog)`) as an embedded application within a program.

The ability to safely execute remote objects as downloadable applications signifcatly extends the functionality of the MBPlayer widgets. For example, the embeddable application contained in the object `<t2-movie-browser>` implements a simple movie browser with the ability to interactively skip to and from arbitrary scene cuts. The embeddable application in the object `<sprog-newsreader>` implements a simple newsreading client. For an illustration of these two embedded applications, see Section 3.4 [Example Application: Media Bank Web Browser], page 35.

3.3.2 HtmlViewer

The HtmlViewer widget provides a hypertext viewer for HTML version 2.0. This viewer supports all standard 2.0 features, with the enhancement that a HTML anchor can refer to a media bank object which will be displayed in the HTML widget in the appropriate locations. The HtmlViewer lacks many of the features provided by modern HTML viewers. In particular, it does not provide any sort of HTML forms support. Nonetheless, it provides a good way to provide a bridge between traditional HTML documents and the media bank by allowing traditional WWW browsers and media bank applications to read the same HTML documents. When a media bank browser reads an HTML document, it can retrieve and interpret the HTML documents as appropriate. When a traditional WWW browser retrieves the same page, it will not interpret the media bank objects directly. Instead, it will attempt to retreive the object through the traditional HTTP mechanisms. At that point, a HTTP transcoder for the media bank object can render a best-effort approximation to the specified media bank object. Such a design allows traditional WWW browsers and the media bank to share the same documents, where the traditional clients will get a best-effort approximation to the document that would be seen by a media bank browser.

3.4 Example Application: Media Bank Web Browser

As an example application of the a dtype-based information system, a simple WWW browser was built on top of the MBPlayer and revised HtmlViewer widgets. This browser differs from traditional WWW browsers both in that it supports the wide variety of multimedia objects provided by the Media Bank, and in that it

permits multiple documents to be viewed at once in context using newspaper-style layout algorithms.

Figure 1 shows the Media Bank browser open to a collection of Media Bank and HTML documents. The upper-left corner contains a simple newsreader that is sent by the server as an embeddable application. The left and right sides of the figure contain a simple movie browser, also sent by the server as an embeddable application. The middle of the screen contains a simple HTML document; the bottom left a simple counting program, also specified as an embeddable application. For example source code to the embeddable newsreader, see Chapter 5 [Appendix A], page 41.

Figure 2 shows the same browser open to a collection of HTML documents. This exmple is intended to demonstrate how the availability of context can enhance the presentation of HTML documents.

File News Sections

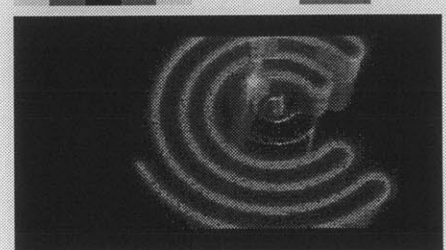
Sat May 27 07:10:06 1995

Page 1 of 2 (1 of 1)

LONDON (Reuter) - Yakwanza, the only surviving gorilla born
by artificial insemination, has at last overcome his aversion to
the opposite sex and impregnated a mate.
Jersey Zoo, in Britain's Channel Islands, said Thursday that
if all goes well, and gestation is the average 260-270 days,
Jersey-born Hlala Kahilli should give birth next January.
Yakwanza, a western lowland

Prev Article 3 of 6 Next

0:00:12



Up 106 Down

Welcome to The Media Lab.

Information

Find out more about The Media Lab and how to join our community. In particular,
The Media Arts and Sciences program is conducting a search for new faculty.

Research

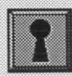
Pointers to research groups and projects in the Lab.

People

- Students
- Faculty
- Staff
- Alumni
- Other affiliates

New and noteworthy

New and especially interesting things to be found on the Media Lab server. Includes links to some of the Lab's online projects.



For Media Lab Members Only

Gene Marquez, a stagehand at Denver's McNichols Sports Arena, prepares the ice Thursday, May 25, 1995, for the opening game of the International Hockey League championship series set for this weekend between the Denver Grizzlies and Kansas City. Comsat Inc., owner of the Denver Nuggets, is trying to wrap up a deal to buy the National Hockey League Quebec Nordiques and bring them to play in McNichols for the upcoming season.

0:07:22

File News Sections

Sat May 27 07:10:56 1995

Related Articles

Page 2 of 2 (1 of 1)

The Media Lab

The Wiesner Building, home of the Media Lab
photo by Steve Mann

MIT's Media Laboratory, founded in 1985, carries on advanced research into a broad range of information technologies including digital television, holographic imaging, computer music, computer vision, electronic publishing, artificial intelligence, human/machine interface design, and education-related technologies. Our charter is to invent and creatively exploit new media for human well-being and individual satisfaction without regard to present-day constraints. We employ supercomputers and extraordinary input/output devices to experiment today with notions that will be commonplace tomorrow. The not-so-hidden agenda is to drive technological inventions and break engineering deadlocks with new perspectives and demanding applications.

SCHOOL OF ARCHITECTURE AND PL

ANNOUNCEMENTS & WHA

(Last Modified May 3,1995)

FORUM PROGRAMS FACULTY ARTICLES PROJECTS CLASSES DEAN'S OFFICE CAMPUS SOURCES

School of Architecture and Planning

The School of Architecture and Planning focuses on the study and design of the human environment – architectural, urban and electronic. Graduate[s] pursue careers in architectural design and tea[ching; in] planning for neighborhoods and cities and rem[ote rural]

FACULTY

Faculty List: MIT School of Architecture and Planning

The School of Architecture and Planning is made up of 117 Faculty members.

- Department of Architecture – 48 total.
- Department of Urban Studies and Planning – 36 total (verify).
- Media Arts and Sciences – 24 total.
- Center for Real Estate – 11 total.

Department of Architecture Faculty – 48 total.

Architectural Design – 28 total

- Julian Beinart
- John DeMonchaux
- Michael Dennis
- Gary Hack
- William J. Mitchell
- William L. Porter
- Jan Wampler
- Atilio Petruccioli
- Andrew Scott
- Roy Strickland
- Julie Dorsey
- Ellen Dunham-Jones
- Takehiko Nagakura

Academic Program

The academic program, also known as The Program in Media Arts and Sciences, is a part of **The School of**

[...] of the school [...]ching; in [...]te rural

4 Conclusion

By providing an easily extensible interface to real-time multimedia primitives, a safe execution environment for foreign programs, and a simple transport mechanism for arbitrary structured data, the Scheme language (and the `dtype` implementation of a Scheme library in particular), make a good basis for a real-time context-sensitive multimedia system. Multimedia objects can contain methods to help control their own interpetation while still allowing the data to be represented in the format most natural for the particular object.

The use of a Scheme interpreter in each client allows servers to send embeddable applications to be executed on the client while still providing for authentication and safety. The `dtype` remote object system and distributed computing environment allows embeddable applications access to a rich computational environment without having to be intelligent in and of themselves. When combined with an appropriate collection of multimedia primitives, the Scheme language in general, and the `dtype` distributed computation environment in particular, provide an expressive and powerful way to support a number of interactive multimedia applications.

4.1 Caveats

The current implementation of the `dsys` and `dtype` libraries suffers from the following limitations: 1) The `pipestream` class still deadlocks sometimes. 2) Neither of the garbage collection schemes are fully functional. 3) The Scheme compiler is still being developed. 4) The exception handling system is disabled because of poor compiler support. 5) The object system has yet to be fully implemented. 6) The dtype protocol currently implemented bears slight differences from the one described. 7) The object namespace servers are not yet implemented.

4.2 Acknowledgements

Much of the original work on the `dsys` and `dtype` libraries was inspired by work done by Nathan Abramson, also of the MIT Media Laboratory, on the `Dsys` and `Dtype` libraries. Although the names and overall design goals of the two systems are the same, they differ fundamentally in both implementation and in design choices. The modern versions can also be distinguished from the original versions by the absence of a capital '`D`' in the name. The names `dsys` and `dtype` are used with the approval of both the MIT Media Laboratory and of Nathan Abramson.

Chris Zimman wrote much of the derived `iostream` library. Josh Cates wrote the multiple precision arithmetic library for `dtype`. Ken Haase wrote the dtype interface to scheme48 and the minimal C dtype library. Ken Haase, Henry Holtzman, Michelle McDonald, Andy Lippman, and Walter Bender all inspired useful features to the `dtype` library.

4.3 References

David L. Tennenhouse *et al*, "The ViewStation: A Software-Intensive Approach to Media Processing and Distribution." Submitted to MM Systems Journal, August 1994.

J. Adam, H. Huoh, *et al*, "A Network Architecture for Distributed Multimedia Systems." 1994 International Conference on Multimedia Computing and Systems. May 1994, Boston MA.

Apple Computer, "QuickTime Documentation." In QuickTime CD, Apple Computer, Cupertino, CA, 1991.

David Gifford *et al*, "An Architecture for Large Scale Information Systems." In Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, Washington, December 1985.

Karen R. Sollins, "Supporting the Information Mesh." In Proc. 3rd IEEE Workshop on Workstation Operations, Miami, FL, April 1992.

Tom Lord, "GNU Remote Operations Web Reference Manual." Publication forthcoming.

Richard Kelsey and Jonathan Rees, "Scheme 48 Reference Manual." Swiss Project, MIT AI Laboratory, 1994.

Erick Gallesio, "STk Reference Manual." Unpublished technical report.

William Clinger and Jonathan Rees, eds., "Revised(4) Report on the Algorithmic Language Scheme." November 1991.

5 Appendix A

The following Scheme code implements the sample embedded application described in Section 3.4 [Example Application: Media Bank Web Browser], page 35.

```
#((uniqueid . "<sprog-newsreader>")
  (type . (item application sprog))
  (function
   .
   (lambda (object context)

     (define w (context-fetch context 'widget))
     (define conn (dsys-connection-create "alphaville" 21002))

     (define avec (vector))
     (define anum 0)

     (define form (xt-create-widget "form" (xt-string->widgetclass "form") w))
     (xt-set-values! form '("x" . 0) '("y" . 0))

     (define article
       (xt-create-managed-widget "article" (xt-string->widgetclass "asciiText") form))
     (define upbutton
       (xt-create-managed-widget "upbutton" (xt-string->widgetclass "command") form))
     (define label
       (xt-create-managed-widget "label" (xt-string->widgetclass "label") form))
     (define downbutton
       (xt-create-managed-widget "downbutton" (xt-string->widgetclass "command") form))

     (xt-set-values! article '("width" . 320) '("height" . 200))

     (xt-set-values!
      downbutton
      '("fromVert" . "article")
      '("foreground" . "black") '("background" . "lightgoldenrod2")
      '("font" . "-adobe-new century schoolbook-medium-r-*-*-*-140-*-*-*-*-*-*")
      '("label" . "Prev"))

     (xt-set-values!
      label
      '("fromHoriz" . "downbutton") '("fromVert" . "article")
      '("foreground" . "black") '("background" . "indian red")
      '("font" . "-adobe-new century schoolbook-medium-r-*-*-*-140-*-*-*-*-*-*"))

     (xt-set-values!
      upbutton
      '("fromHoriz" . "label") '("fromVert" . "article")
      '("foreground" . "black") '("background" . "lightgoldenrod2")
      '("font" . "-adobe-new century schoolbook-medium-r-*-*-*-140-*-*-*-*-*-*")
      '("label" . "Next"))

     (xt-manage-child! form)
```

```
(define set-anum!
  (lambda (x)
    (if (< x 0) (set! x 0))
    (if (> x (- (vector-length avec) 1)) (set! x (- (vector-length avec) 1)))
    (define s (if (< x 0) "" (vector-ref avec x)))
    (xt-set-values!
     label
     (cons "label" (stringout "Article " (+ x 1) " of " (vector-length avec))))
    (xt-set-values! article (cons "string" s))
    (set! anum x)))

(set-anum! 0)
(xt-add-callback! upbutton "callback" (lambda () (set-anum! (+ anum 1))))
(xt-add-callback! downbutton "callback" (lambda () (set-anum! (- anum 1))))

(define update
  (lambda (t)
    (cond ((equal? (dsys-station-poll conn) 'pending)
           (define d (dsys-connection-read conn))
           (define s (vector-lookup d 'body))
           (vector-append! avec s)
           (if (>= anum 0) (set-anum! anum) (set-anum! 0))))))

(lambda (sym . args)
  (cond
   ((equal? sym 'create) '(#t . 0))
   ((equal? sym 'destroy) #f)
   ((equal? sym 'update) (update 0) '(#t . 0))
   (else (exception "invalid method"))))))
```